NTNU – Trondheim
Norwegian University of
Science and Technology

Conversion of a simple Processor to asynchronous Logic

Bjørn Thomas Søreng Vee

Electronics System Design and Innovation
Submission date: June 2014
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Ronan Barzic, Atmel

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

1 Summary

This paper discuss the conversion of a simple 16-bit synchronous RISC based processor into asynchronous logic. The most important targets were the simplicity of the conversion, to see how the tools reacted to asynchronous elements, increase the stability of the processor under different conditions and make some general guide lines for conversion other processors.

The report first gives a short introduction to design of asynchronous logic. Then it discuss approaches for the conversion before the actual implementations done are walked through. The area, power and performance for different implementations is also discussed.

3 different asynchronous implementations was implemented and tested: One simple sequential request-acknowledge scheme version with no pipelining, one simple muller pipeline based implementation and last a more advanced muller pipeline based version with more pipeline steps, FIFOs to avoid hazards and register feedback.

The simple pipelined muller version is verified in fpga as well as the original synchronous version.

The main challenge was to decide how the conversion should be done, and get the synthesis tool to synthesize the design correctly. We got the asynchronous versions of the processor to work after synthesis by adding constraints on paths that exploit propagation delays to avoid the synthesis tool from optimizing those parts of the circuit. During the synthesis and constraint generation it was necessary to analyse the synthesis output to verify that the generated netlist was as expected.

The fpga implemented asynchronous processor is not optimized for performance, but more for testability: We have added programmable delays and the possibility to control the processor with both a clock or the synchronous controller. A weakness with the report is that we have not had time to manufacture an actual asic to run and compare simulations against a real asic.

Both rtl simulations, ntl simulations and fpga tests show that the implemented designs work as expected.

We have not extracted exact timings from the actual routed asic design as the routing of the design is still a work in progress, it is however simple to see that for each extra pipeline step on the critical path of the processor, the area will increase, and the processor will be slowed down and the power will increase. This is because our processor is very sequential and thus the gain from more pipelining is zero. Still, it was important to see how more advanced implementations was handled by the tools. As the processor is small (if we look away from the ram blocks), the relative increase in the area

and power for each pipeline stage is large.

A closer look into dynamic delay scaling, models for signal propagation under different conditions and timing assumptions would be a natural way to continue the future work.

2 Preface

Micro electronics are becoming smaller and smaller. At the same time, we use more and more portable electronics and the technology to harvest energy from the environment is improving. One of the bottlenecks today is the power consumption/performance ratio. Asynchronous logic can both lower the power consumption for microelectronics and improve the performance. Asynchronous pipelines was popular experimented with in the late 80's and 90's. One of the problems was the complexity and limitations in the tools when it came to asynchronous implementations.

I chose this project because I wanted experience working in a large electronics company. I hoped to gain more experience using state of the art tools typically used when designing electronic circuits. I have always enjoyed practical work, learning by doing and hands on experience. In this project I had the opportunity to implement a physical system, and verify functionality of the system.

The project was started by making a time plan for the project. You find the plan in appendix D. The project went as planned, even a little bit ahead of schedule so we managed to start a the asic layout for the synchronous processor and fpga implementation for the synchronous processor and one of the asynchronous processors.

During the project I have gained more experience using different tools for designing electronic circuits. I had no experience using a typical tool flow with scripts and autogeneration. I had never used Verilog before and I have gotten more familiar with Xilinx FPGA, USB, VGA, and Synopsys tools.

I would like to thank the people that assisted me in the project work: The people at Atmel, specifically Ronan Barzic (project supervisor). I would also like to thank Kjetil Svarstad (project supervisor at NTNU) for assisting me through the project.

Contents

List of Figures

3 Introduction

This paper discuss the conversion of a simple synchronous 16-bit RISC based cpu to asynchronous logic. The targets were making a processor that is more stable under different operation conditions, on using a tool flow that is used for synchronous design, use the tools in a generic way that can be used for conversion of other synchronous processors and the simplicity of the conversion. In addition to this, the most basic performance aspect is evaluated and discussed to get some quantized data as well.

By testing several different implementations with different typical asynchronous elements we have been able to see how the tools react to different challenges in asynchronous logic.

Asynchronous logic was popular in the early 90's but designing was difficult due to limitations in the tools. However, new application areas for CPUs require more and more power efficient circuits to operate under a lot of different conditions. Asynchronous design can both increase performance and lower the power consumption. More and more microelectronics are also connected to the internet. The internet of things is demanding even more power efficient and stable ways to communicate globally with units that can handle a ton of operating conditions.

The work was stared by a literature study about asynchronous designs. A basic theory about asynchronous design is given in chapter 4. Then a study of the synchronous processor was done and the tool flow was analysed in chapter 5. After this, different conversion strategies for the design was analysed in chapter 6. The implementations, how they were done and challenges we met during the implementations are discussed in chapter 7. Then the results from the implementations are listed in chapter 8 and discussed in chapter 9 before a conclusion is given in chapter 10.

Note that some parts of the flow is classified, but not necessary for understanding or verification of the results as this is mostly scripts that automates the steps in the flow as described in chapter 5.1. All cpu sources and constraint files used in the different steps are included in the appendix and in the attached zip archive.

The report is partitioned as follows:

- Ch. 1 - A summary of the project.
- Ch. 2 - Preface
- Ch. 3 - Introduction and problem definition
- Ch. 4 - Background and theory. Important theory and references to more information are given.

- Ch. 5 - Previous work walks through the starting point of the thesis: We walk through the synchronous processor and the flow used for implementing the processor.
- Ch. 6 - Architecture specification and design - Different implementation options and strategies are discussed and evaluated.
- Ch. 7 - Implementation and verification - The different implementations done are discussed and elaborated before the asynchronous challenges on each step in the flow is discussed.
- Ch. 8 - Results lists results when it comes to simulation, performance, synthesis and physical testing.
- Ch. 9 - Discussion discuss the results and try to give some general guide lines to convert synchronous to asynchronous design.
- Ch 10 - Conclusion. A conclusion and ideas for future work is given.
- Referances and Appendixes contains constraint files, verilog sources, time schedule, schematics etc.

3.1 Problem definition

[10, Starting from an existing synchronous RTL implementation (in verilog,) of a very simple processor, derive an asynchronous implementation following the micropipeline (or equivalent) paradigm. First, the student will study the existing synchronous implementation to get familiar with the architecture and the simulation tools. Then following literature examples, he will derive one or several asynchronous implementations that will be verified using simulation and eventually on a Xilinx FPGA. The student will have to investigate how to generate the control part (by "hand" or by using existing tools) The required software, hardware and a working place at Atmel's office could be provided.]

Other specifications given from Atmel after project start were high level and are listed under:

- Atmel wants focus on using a good tool flow that can be reused. A flow for the synchronous processor is given, and the asynchronous flow should be as close as possible. The target for the project is to design a chip where stability under different conditions can be tested, but performance may also be evaluated.

- How the design is implemented when it comes to for example communication and storage elements is not important.
- The student should try to give some general guidelines to convert synchronous to asynchronous design with the given flow.
- A good way to test and verify the design is important.
- The processor might be used for low power.
- If time, FPGA and/or asic implementation is a goal

4 Background and theory

In this section we will look at typical approaches to implement asynchronous logic and communication. We have also looked at other processor implementations. Some notes about them are included in appendix I. This will help us later to make design chooses. Most of the theory is based on [1] chapter 2 and [7].

4.1 Clocking vs handshaking

The main difference between synchronous and asynchronous logic is the clock. While synchronous logic uses the clock as synchronization, asynchronous logic uses handshaking. The removal of the clock tree and the clock buffers usually saves power (less switching) and can increase the speed because we do not have to use the worst case propagation delay of the circuit as the clock period.

4.2 Handshaking protocols

We typically divide the asynchronous handshaking protocols into bundled data and one-of-n channel.

4.2.1 Bundled data

Bundled data protocol means a single request line bundled with a unidirectional single rail data bus. It also has an acknowledge line as shown in figure 1 to signal that the data is accepted.

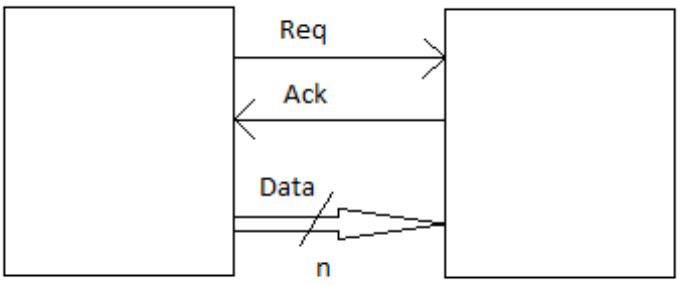


Figure 1: Bundled data (push channel)

A channel can be pulled or pushed. If pushed, the request is in the same direction as the data path. If pulled the request goes in the opposite direction

of the data. Bundled data push channel normally leads to the most efficient circuits because of the possibility to use timing assumptions to minimize delays.

Bundled data channels are also more energy efficient (for wide channels) and smaller then for example a dual rail implementation. It is also the solution that is closest to typical synchronous channels. The disadvantage of bundled data channels is the timing assumptions and that all data wires related to the request signal has to be ready when the request is generated.

[7] state that an disadvantage is that there is no clock that can be slowed down if operation conditions are changing for the worse compared to a synchronous implementation with dynamic clock scaling, however by using a programmable delay line and dynamic delay scaling this disadvantage can be removed and the margins on the forward latency path can be even better then a synchronous implementation for pipelined designs.

4.2.2 4-phase (return to zero) bundled data

In the four phase protocol, only the transition from low to high on the request/acknowledge lines signals an event (or only transitions to low). The principle is shown in figure 2.

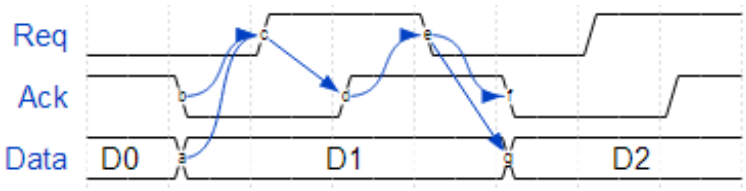


Figure 2: 4-phase protocol

The advantage of this is that you have more defined states then for example the two phase protocol, the controllers are more simple and the request and acknowledge signals may be used directly as a clock. The disadvantages is the superfluorus return to zero that cost both extra time and energy.

4.2.3 2-phase bundled data push channel

In the two phase protocol each transition on request/acknowledge wires signals an event. The principle is shown in figure 3.

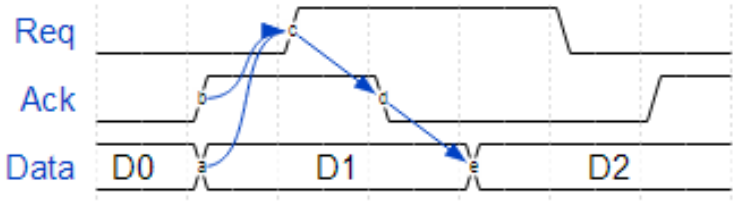
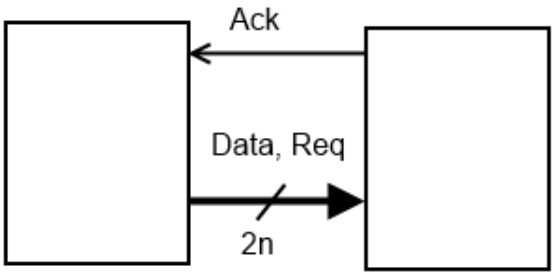


Figure 3: 2-phase protocol

The advantage for the two phase protocol compared to the 4-phase protocol is that it is faster and more power efficient. The disadvantages is that it is more complex to design and that responding to events are more complex.

4.2.4 Dual rail protocol

The dual rail push channel protocol is rooted in David Muller's work in the 1950s. The dual rail protocol is shown in 4. Here the request lines are individual, as every data bit is coded with 2 bits.



	d.t	d.f
Empty	0	0
Valid 0	0	1
Valid 1	1	0
Not used	1	1

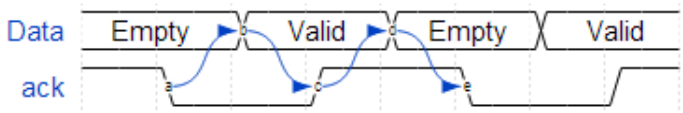


Figure 4: 4-phase dual rail protocol example

The dual rail protocol leads to larger and more power consuming circuits, but have advantages in some cases (stability due to the return to zero before every data and individual data bit propagation).

4.2.5 Other protocols

We have almost infinite possibilities for implementation of asynchronous logic and communication protocols depending on the usage.

The 1-of-n channel protocols uses N data wires to sent log2 N bits of data. Typically with four phases.

1-of-2 is the same as dual rail.

1-of N single track - is a 2 phase protocol. It offers lower power as there are fewer transitions per bit compared to 1-of-n channel. Still the circuits often has larger power consumption then standard bundled data protocols.

Figure 5 shows an example of the 1-of-4 protocol.

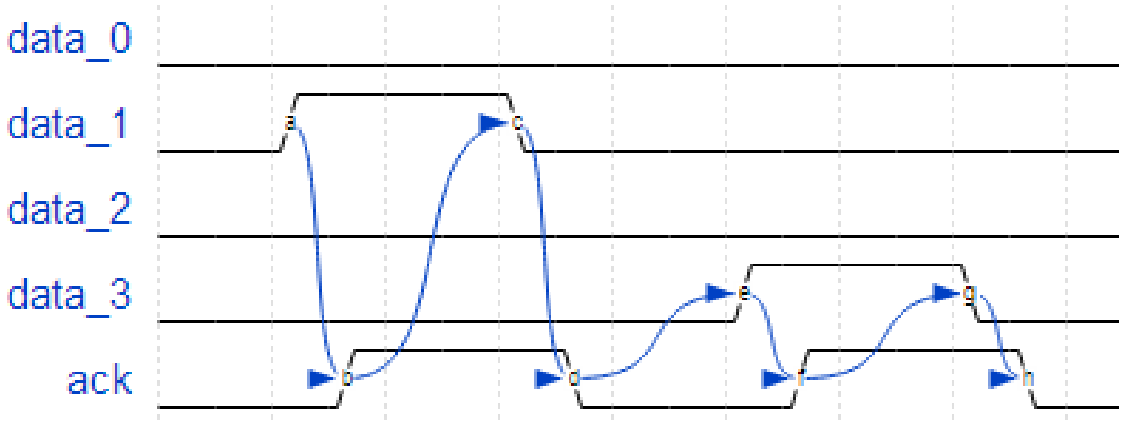


Figure 5: 1-of-4 channel example

4.2.6 Conclusion - handshaking protocols

We have no general rule which protocol which is the best. It depends on the specification, the designer and the usage. However, most practical implementations use one of the protocols mentioned or simple modifications of them.

4.3 Pipelined handshaking

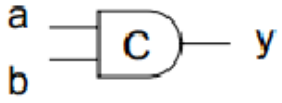
Pipelined handshaking can improve performance, as multiple instances can operate concurrently. You can read more in: [7] chap. 2.2.2. Pipelined handshaking can be compared it to a fork and join operation.

4.4 Implementation of asynchronous pipelines

The Muller pipeline approach is maybe the most famous implementation of an asynchronous pipeline. It originally used the two phase protocol, but it is simple to modify to a four-phase protocol pipeline. See [7] chapter 8 and 9 for more detailed description.

The muller pipeline was first described in [2] by Ivan Sutherland. The muller pipeline uses so called muller c-elements to control the request acknowledge signals and [2] also describes other elements that can be used to fork, join, mux, toggle and other different operations that might be needed for the asynchronous controller. Figure 6 describes the muller c-element. Figure 7 shows a two phase muller pipeline with processing elements.

4.4.1 The Muller C-element



a	b	y
0	0	0
0	1	No change
1	0	No change
1	1	1

Figure 6: The Muller C-element

4.4.2 Muller pipeline

A two phase muller pipeline with processing is shown in figure 7. For a four phase verion, the capture pass latches is replaced with simple enable latches. Enable is connected the same way as the input port C in the capture pass latches. The P port is removed. We can also have a pipeline without processing. In this case, the delay and comb block is removed.

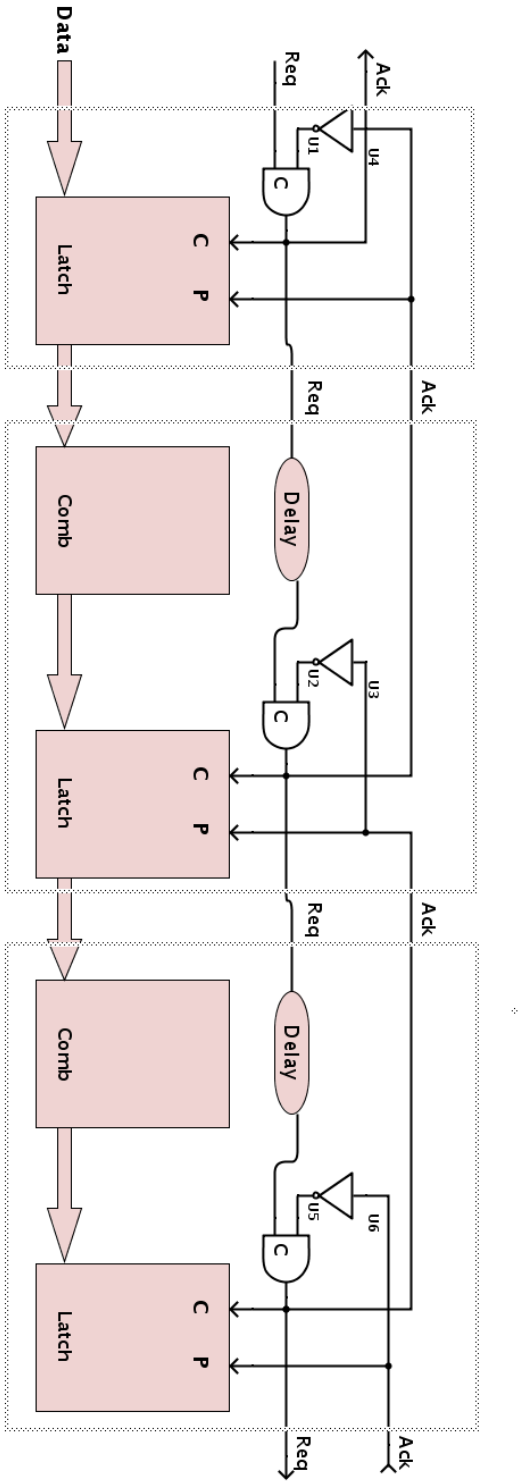


Figure 7: The Muller pipeline with processing 2 phase

4.5 Delay lines

Delay lines are used to control the timing so the transitions between the stages of the pipeline are done when the data is valid. This is typically done by making chains of logic elements that has a propagation delay longer then the combinatorial logic in that stage. More in: [7] chapter 9.4

4.6 Speed

The speed of the circuit is measured the same way as for synchronous circuits. You analyse the propagation delay in a given path. The difference is that the "clock" is the request signal and that we have several different clocks. Also, the actual performance is not dependent on the processing logic path, but rather on the delays on the delay lines chosen. It is therefore important to use a delay that matches the delay in the combinatoric logic in the delay lines. You can find more information about performance analysis in [7] chapter 5.

4.7 Fork/join

Fork and join can be used to do several operations in parallel on the pipeline. Forking is done by splitting the request signal, joining can be done with a muller c element.

4.8 Muxes

Muxes can be used to select paths for the request/acknowledge signal depending the instruction. This way you do not have to wait the worst case propagation delay for a pipeline stage.

4.9 Pipeline/bubbles

For pipelined designs, bubbles might be necessary to avoid hazards during runtime. Bubbles can be done the same way for asynchronous and synchronous design. You can for example stall the processor or run other instructions until the the needed data is available.

4.10 ALU optimizations

One of the advantages from asynchronous designs is that you can have separate delays for different paths in the design (for example depending on the instruction). This way you can always have the fastest circuit possible at the

cost of power. This can be done with separate delays for each path (area overhead) and muxes or programmable delay elements and dynamic delay scaling (complexity increase).

4.11 Earlier implementations

You will find some notes on earlier implemented asynchronous processors in appendix I.

5 Previous work

The synchronous processor used as a starting point in this thesis is modular when it comes to instructions due to an automated flow to add and remove instructions. The processor implements the instructions described in [8] (You find this file on the root of the .zip archive). Just a few of these instructions was tested with rtl simulation before the work with this thesis stared. The initial veriog source code for the processor fully implements a decoder, program ram, register, an alu and muxes to mux the signals to the correct component. The processor is implemented in one large verilog module except for the program ram and the registers that are individual modules. Figure 8 shows a simplification of the cpu. You can find a more detailed figure in appendix F. Figure 8 shows the clock tree of the synchronous processor. The processor has a simple pipeline with instruction fetch, execution and write back. An additional step with memory access is added when we have load-/store instructions. A few fixes to the synchronous processor was needed to make it synthesizable. You find these changes in appendix J. Basically the data ram was added and a delay slot was removed.

Figure 8: Clock tree synchronous CPU

5.1 Work flow

In this subsection we will walk through the design flow (shown in figure 9) of the synchronous cpu and show in what steps we needed changes to successfully implement the asynchronous design. For more details about the changes, see chapter 7.

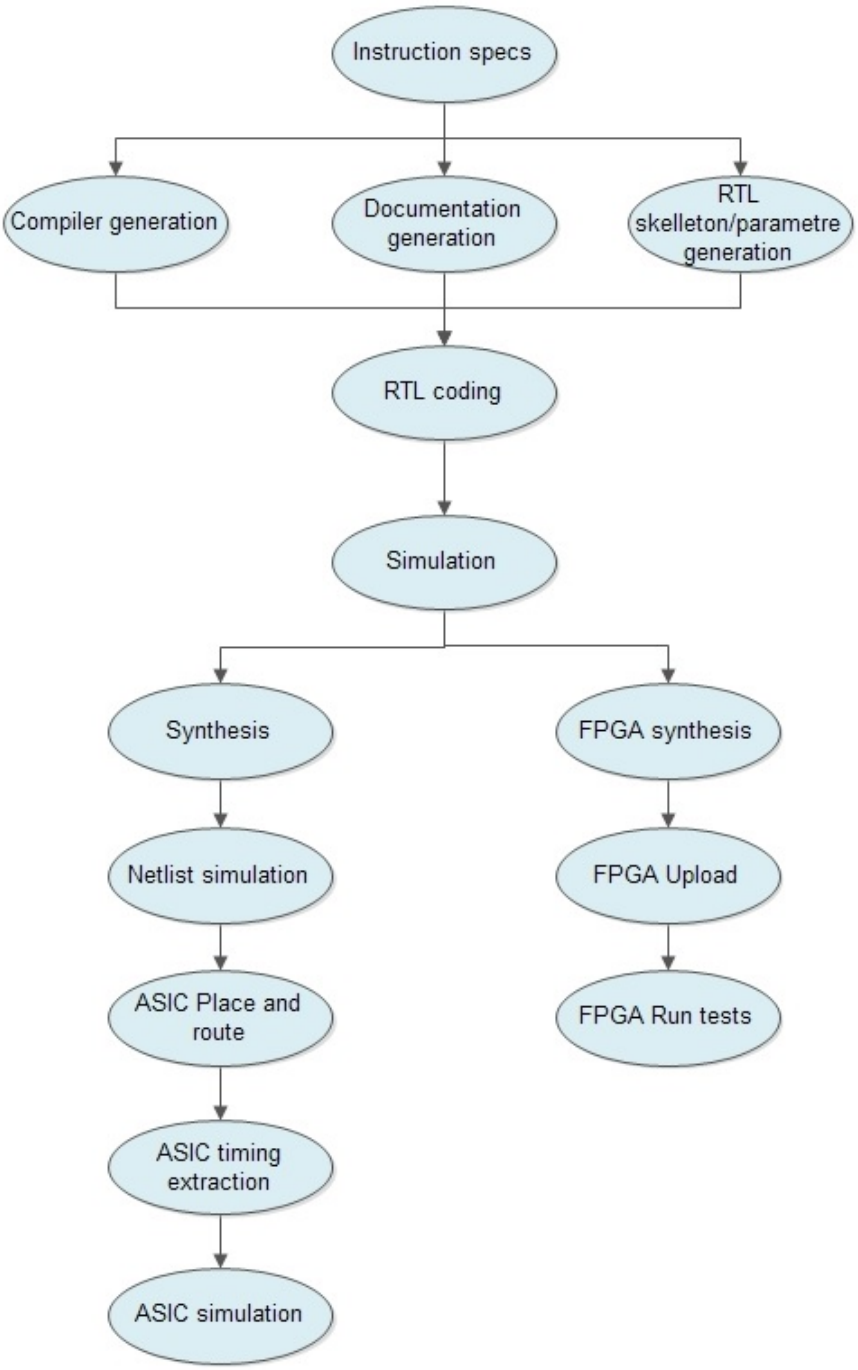


Figure 9: Design flow overview

5.1.1 Documentation, compiler and verilog skelleton generation

.xml files with instruction specification is written and compiled into docbook files. The instruction specifications are extracted and given as input to a SystemC archc[11] project. This is compiled with archc. the output is a gnu compiler. This compiler can generate test programs from asm. Also, verilog parameter files, the decoder and instruction muxes are autogenerated from the instruction files. This part of the flow was exactly the same for the asynchronous processor and the details is not interesting for this thesis.

5.1.2 RTL coding

The functionality of the processor is manually written in verilog. Rtl changes was needed for the asynchronous processor.

5.1.3 RTL Simulation

The processor is simulated with a verilog testbench that loads a program memory. Test programs are written in asm and compiled into program memory vectors with an archc compiler specific for this processor. Small changes was needed for asynchronous processor due to a different communication interface. An example of the test bench is given in appendix K.D and K.E. You find a test program example in appendix K.F and K.G.

5.1.4 Synthesis - ASIC

After successful simulation, synthesis is done with Design Compiler. The output of the synthesis is often not simulated as the simulation might give strange results as no propagation timings are added during synthesis, but we have decided to do a netlist simulation for this project to be able to find errors in the asynchronous design after the synthesis optimizations more simply.

To do synthesis of the asynchronous design, constraints was needed to be added to the synthesis. The constraints are more discussed in chapter 7.4.

5.1.5 NTL Simulation

Simulation is done with VCS. We could use the same test bench and test programs as in the rtl simulation for both processors, but needed some changes for checking the registers at the end of the simulation as all registers are merged into one large vector in the netlist design. These changes was only needed due to the use register auto test check scripts in the Atmel flow.

5.1.6 Place and route - ASIC

Place and route was done for synchronous processor. We expect that no changes are needed in the flow for the asynchronous processor, but this is not tested. The layout is done in 130 nm process with standard cells.

5.1.7 Synthesis, place and route FPGA

For the fpga implementation a flow using the Xilinx ISE GUI is used for this project. It was decided not to make automated scripts for this part, as the gui was the safest and fastest way to be sure that the results from the fpga synthesis is not disturbed because of errors in tcl scripts (eliminate sources for error). Xilinx ISE can autogenerate automated tcl scripts for the flow after it is done in the gui. None of the processors was synthesised for fpga before the project started, but the flow for synchronous and asynchronous processor was the same, except for the extra keep constraints for the asynchronous processor.

5.1.8 Physical testing

The physical testing for this project was done by generating a flow using libfpgalink library to communicate with the fpga over usb. This is described in chapter 7.8. We have no difference between the synchronous and asynchronous processor related to the communication during testing, except more configuration options in the asynchronous cpu.

6 Architecture specification and design

In this section we have looked at different important properties that helped us decide how we should convert the processor. We focus mostly on these properties: the simplicity to convert the design, testability and last we look at the performance (area,power,speed).

As this is a really simple synchronous processor, as shown in figure 8 and appendix F, the gain from pipelining and speed optimizations that could be done will in many cases slow the processor down and increase the power consumption a lot. This will also be discussed more for each implementation in chapter 7.1.

6.1 Design choices

We will now discuss some of the design choices made.

6.1.1 2 phase, 4 phase or dual rail

As discussed in chapter 4 the two phase protocol has some advantages over the four phase protocol: The speed is higher and the power consumption is lower if you make a good design. If we had used the four phase protocol, we could have used the ack-lines directly as clocks for single edge memory elements. The dual rail protocol would increase the power consumption and does not add any additional challenges related to the asynchronous synthesis compared to the two phase protocol. Since the two phase protocol differs the most from a normal synchronous circuit it was decided that the two-phase protocol would be the most interesting solution to implement.

6.1.2 Flipflops with pulse generators, dual edge flipflops or latches

Since we used a two phase protocol we had to decide if we wanted to use dual edge flip flops, latches or single edge flip flops with a pulse generator. Atmel has single edge ram that would be suitable for this processor, so we would need some pulse generators in our design. In the beginning we used dual edge flip flops, but the synthesis flow did not support this without adding a lot of control logic. Single edge flip flops with pulse generators and event controlled latches are therefore used in the synthesized designs. The pulse generator design is shown in figure 10.

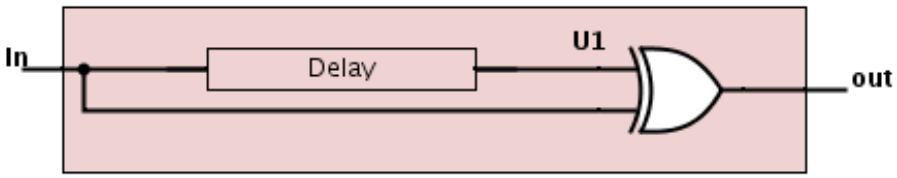


Figure 10: Pulse generator

For register forwarding we wanted to use registers for storing data. This is due to a more simple synchronization between neighbouring stages on the pipeline. Both event controlled latches and registers are tested in synthesis. With event controlled latches is difficult to predict if it is in capture or pass mode at a given time. Therefore forwarding from a simple event controlled latch would not work without several timing assumptions. In registers we know the value is stored until a new cycle, however you will probably add penalty due to propagation delays (dependent on the synthesis implementation of the gates). Flip flops are usually larger and slower then event controlled latches. By using flip flops, the core of the cpu also needs less changes and only the clock tree needs to be changed into asynchronous control logic to get an operational asynchronous cpu.

6.1.3 Delay line design

The delay line design was done by adding chains of logic elements on the request and acknowledge lines. For testability and for later functionality (for example dynamic delay scaling) it was decided to make the delays programmable by adding a mux and selecting one of the output wires on the chain. Unfortunately we had no time exploring dynamic delay scaling in this thesis. Figure 11 shows the delay chain designed. Note that U1, U2, U3 and U4 actually need to be two not gates for correct functionality but is skipped in the figure to save space.

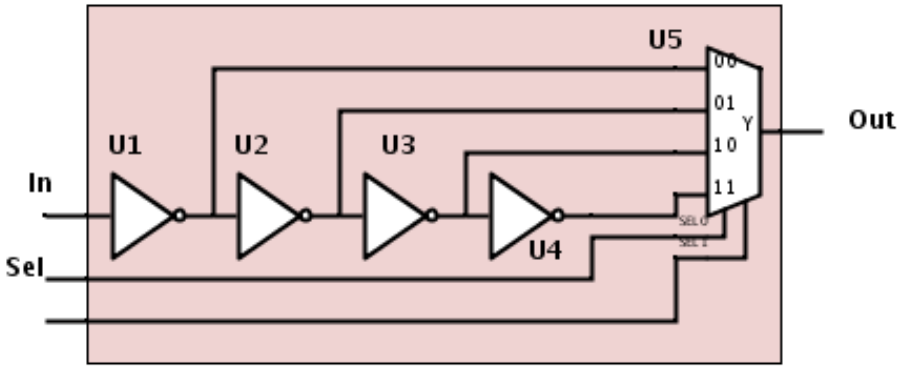


Figure 11: Delay chain module

6.1.4 Pipeline implementation

It exists several design references for asynchronous pipelines. We chose to base our design on the muller pipeline as this is the most known, and will give our tools a typical asynchronous challenge. Also Atmel has muller c modules ready. The Mouse trap pipeline is an example of a faster pipeline, but it adds no asynchronous or complexity challenges compared to the muller pipeline.

6.1.5 Possibility to control the asynchronous cpu with a clock

We wanted to be able to debug the cpu with a standard clock. Therefore muxes and some control logic was added to the asynchronous design that is implemented in fpga and asic to be able to select between a clock or the asynchronous controller. This also makes process of timing extraction from the design closer to a normal synchronous processor.

6.1.6 RAM blocks

Initially simple verilog ram blocks made for this processor was used, but when preparing for asic implementation, the simple ram was changed to standard cell ram. Simple memory wrappers was made to control the two ram blocks. The standard cell ram used is a 32 bit ram with 1024 addresses. Since our processor has the possibility to address 16 bit addresses 16 bit data, a wrapper with decoder was created to use all the bits available in the ram effectively.

For the fpga we used the ram blocks inside the fpga by using Xilinx core generator.

6.2 Conclusion

Since it was more important for us to see how the tools react to different asynchronous design elements and testability rather then the fastest and smallest processor, we have decided to do several asynchronous implementations in parallel. We have focused on implementing designs that we thought would give us the most challenges in synthesis and lay out compared to the standard synchronous design. We used the 2 phase protocol and made different designs with delay cells, pulse generators, event controlled latches, fifo, clocked flip flops, register forwarding and muxes on the request/acknowledge lines.

7 Implementation and verification

In this chapter we will first go through the implemented processors. Then we will go through the common challenges we met through the flow and how they were solved.

7.1 Implemented processors

We decided to work with 4 different versions of the processor to be able to compare results when it comes to performance, speed area and difficulties in synthesis for different asynchronous elements. Two of the processors are also implemented and verified on fpga. Table 1 shows the progression for the different processor cores.

Table 1: Progression

Progress	Synchronous	Ring req ack	simple pipe	adv pipe
RTL similation	X	X	X	X
Synthesis	X		X	X
NTL simulation	X		X	X
FPGA	X		X	
Place and route	X			

7.1.1 Synchronous

The original synchronous processor had two delay slots, some bugs and not implemented the data ram. The critical bugs was fixed as explained in appendix J. One delay slot was removed and the bugs was fixed. The data memory was added. We made one where the ram access is done on negative clock edge and one where we stall the processor for one cycle on data memory access. This is selected by defines on compile time. We will have the best performance by stalling, as the ram is the slowest module in the cpu and if using the negative edge of the clock we have to decrease the maximum clock frequency that will give us a penalty every clock cycle.

7.1.2 Asynchronous - Ring req, ack

As a starting point it was decided to create a really simple asynchronous version based on a simple request acknowledge scheme to see how the compilers and simulation tools reacted, get familiar with the architecture of the

processor, the tools and verilog. Figure 12 shows the initial asynchronous processor implementation. The simple request acknowledge scheme was implemented with both for 4 phase protocol, and for 2 phase protocol. The 2 phase protocol has added pulse generators on the register file and ram to be able to reuse positive edge triggered modules.

This solution was very simple to implement as every module is requested sequentially and the first module waits for the last module to finish before generating a new request for a new instruction. A problem with this implementation is that it will be slower then the synchronous implementation due to no pipelining, it does not work exactly as the pipelined synchronous version and we do not get to see how the synthesis tool and place and route tools react to more advanced solutions.

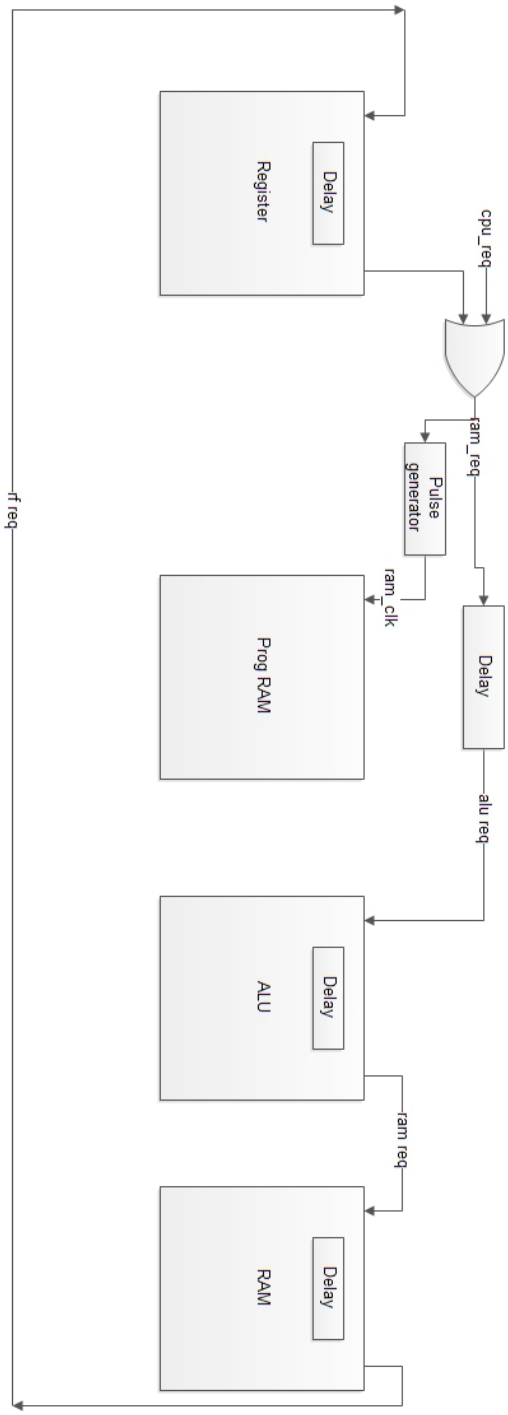


Figure 12: Ring request scheme

7.1.3 Asynchronous simple muller pipeline with posedge flipflops

This is the implementation that is the closest to the synchronous processor. It has 1 delay slot (depending on when we request a new instruction) and the same pipelining as the synchronous processor. Because of this we can look at the processor core as a black box when testing and only look that the input and outputs are the same for both processors for the same tests. This is the processor we have spent the most time testing.

The pipeline stages are controlled by using a muller pipeline based scheme. For testability it has programmable delay lines and the possibility to use a standard clock by muxing a clock and the asynchronous controller as shown in figure 13. The design uses positive edge flipflops for storage, so the conversion from synchronous to asynchronous is basicly only done on the clock tree.

Notice that if we wait for the alu to finish if we have a branch instruction before fetching a new instruction to have no delay slots on branching (and avoid hazards) for the cpu, as illustrated in figure 25 (in appendix J.H). This would cause the processor to be a bit slower then the synchronous version but more stable as the program designer does not have to think about hazards related to the PC counter jump branch instructions. This is a simple modification to do.

This processor is the implementation we have focused on in the physical implementation and implemented on fpga. Figure 13 shows the implemented processor.

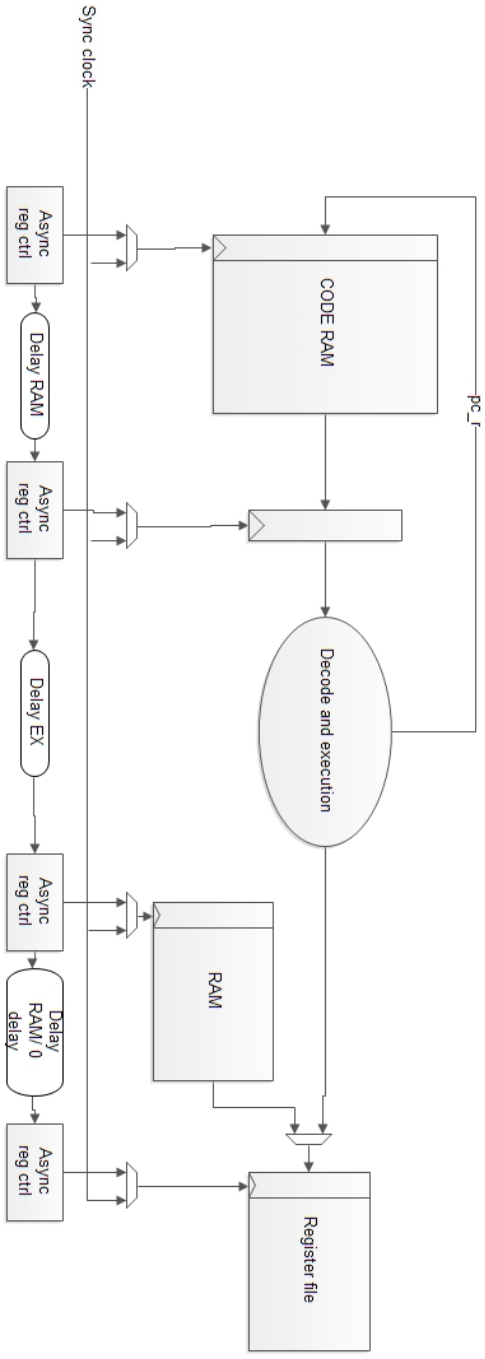


Figure 13: Schematics dualclock

7.1.4 Asynchronous advanced muller pipeline with event controlled latches and FIFOs

See appendix G for a schematic of the multistaged CPU. Note that the FIFOS are not included in the schematics.

To be able to test the tools properly a more advanced muller pipeline implementation with register feedback, event controlled latches, FIFOs and stalling (shown in apppendix G) was implemented and tested with the flow.

In this design we have used event controlled latches between each stage, and flipflops to store values for forwarding. We fetch a new instruction as fast as the instruction is captured in the decoder, as long as the instruction is not a branch instruction. If we have a branch instruction or hazards due to data dependency , we stall and/or forward the needed data to the alu as soon as possible. The FIFO checks we used for this implementation is not optimal as it does not use any timing assumptions, but they are generic, which means it can be used without thinking about the delays for each element in the pipeline.

When delays are extracted from the place and route tools, several optimizations would be possible (stalling could be based on time and not the number of instructions since the dependency).

There are of course other and maybe smarter ways to convert the processor, but since we are not that interested in the fastest or most power efficient cpu, comparing what happens in the synthesis and place and route tool, comparing complexity and area of these solutions we will get a good ground pillar to make design decisions in the future with these tools.

We will try to describe the FIFO protocol: Each time a new instruction is decoded, the destination, pc counter and operation is stored in a FIFO. The FIFO is as long as the pipeline. If an instruction has a destination that matches one that is stored in the destination FIFO, and the CPU is not stalling we check the operation. If the operation with the same destination is not a load/store (LW/SW/LB/SB) operation, we forward the alu result if the instruction with the matching destination was the last instruction. If not, we flush the pipeline until we are sure the result is propagated to the register file. The flushing is done by not incrementing the pc counter, sending new RAM requests and disabling the register file clock.

7.2 Structural change

It was decided to keep the structure from the synchronous processor for simplicity and add asynchronous modules to the main processor core file: simd_useq.v

7.3 RTL design

RTL design was done in verilog. Compilation was done with icarusverilog at the beginning, but some of the components from the standard cell library was not supported. (setupandhold). Therefore VCS 2013.06_64 was used for compilation and simulation.

Gtkwave was used for plotting the dump files from the simulation.

7.3.1 Asynchronous challenges

The main difference in the RTL design was that we exploited propagation delays. This made the design a bit more complex, and as we discuss in chapter 7.4.1, the parts of the design that exploit propagation delays is optimized away by the synthesis tool.

A good rule is to keep the request/acknowledge wires as clean as possible to aviod strange ripple and switching effects.

Properly resetting the pipeline on start was also important and often a source of error when we saw strange behaviour.

7.4 Synthesis - ASIC

For synthesis, Synopsys design compiler 2013 SP2 was used.

7.4.1 Asynchronous challenges

Synthesis was problematic for dual edge flipflops with the Atmel flow and standard cells. It was therefore decided to add pulse generators to the design to generate a clock pulse for flip flops when there was an acknowledge for the positive edge controlled modules. If we had used a four phase protocol, we could have used the ack signal directly as a clock.

The synthesis tool optimized away modules that exploited propagation delays. For our design this meant pulse generators and delay lines. Often this lead to a simple design where the synthesis tool was able to optimize away most of the cpu.

The problem with optimization of modules that exploit propagation delays was solved by using set dont_touch constraints (for Synopsys Design Vision) on the correct nets and modules. For out case this was on request and acknowledge signals, the output of the pulse generators and the output of the delay modules. We also had to set dont_touch on the output inside each of the modules that exploit propagation delays.

As an example, figure 14 shows a typical delay module inside another module (Note that U1,U2,U3 and U4 represents two not gates for correct

operation). Here we would have to put a keep on net done,out2 and out1 to avoid the synthesistool to optimize away the delay functionality. If protecting the whole module, the synthesis tool would simply keep the delay chain, but route the ack net directly to the done net. Also when keeping whole modules the synthesis tool did not use the correct modules from the standard cell library: The modules was converted from verilog to GTECH standard cells, but not to the Atmel standard cells. When protecting only out2, In was routed directly to out2.

It was necessary after a synthesis to open the netlist schematics to verify and debug the design arround modules that exploit propagation delays when the netlist simulation gave unexpected results.

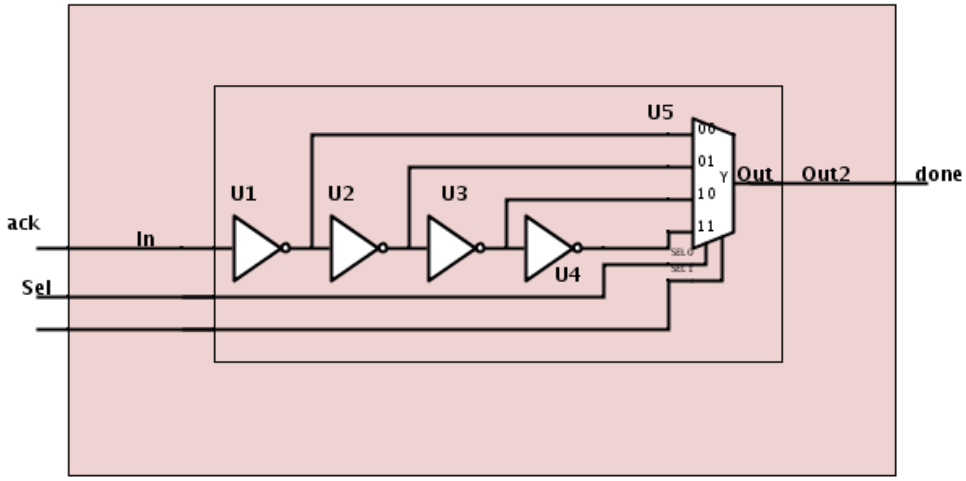


Figure 14: Delay cell inside module

The latches in the design synthesised as expected. You find the constraint file used in appendix B.

7.5 Design analysis

We wanted to extract timing, area and power from our designs. Area was directly reported from the synthesis tool. To extract timing and power, Synopsys prime time was used. We only had time to do very basic analysis. This is discussed more in chapter 9.

7.5.1 Asynchronous challenges

The worst case path of the design is of course the delay lines. We used set_disable_timing constraint on the delay modules (RAMD, ALUD, MEMD, RFD) to skip the delay lines from the analysis.

For our dual clock design we also used setcaseanalysis constraint on the clock selector to tell the tool that the analysis should be done with a standard clock on the muxes. This way, prime time will look at the timing for each stage where the clock is muxed in. Timing extraction is possible directly on the asynchronous design, but it is necessary with good constraints and to define the paths you want to analyse. The dual clock scheme makes the process more simple. It will also be possible to extract timings from each stage on the asic by applying one clock pulse and see at what delays the results from each stage fails in a simple intuitive way.

7.6 Synthesis - FPGA

For fpga Synthesis Xilinx ISE was used. A development board from Digilent called nexys 2 with spartan-3E 1200k FPGA was used. The problems discussed in Synthesis - ASIC (chapter 7.4) was the same for the xilinx ISE synthesis tool. For the constraints, an .xcf file was added with keep_hierarchy and keep constraints. You find the constraint file in appendix C. In xilinx ISE it was possible to keep a net on every module of one type independent of the name of the module. This was an advantage and made the constraint file generation more simple.

7.6.1 Asynchronous challenges

When Protecting the same nets as in the asic synthesis we met no additional challenges. There was some problems with some of the syntax. c2log was not supported (was worked arround by making a c2log function) and every index of 2D arrays needed to be in the sensitivity list individually when reading/writing.

7.7 Verification

As the synchronous processor was not tested that much, the first step in the verification process was to generate more test programs for the processor. As the asynchronous processor was supposed work exactly the same way as the synchronous processor except for no clock, the tests could be used for both processors.

An example asm test file is included in appendix K.T. This file is compiled with a GNU archc compiler and results in a file as shown in appendix K.U. One of the problems with this test methodology is that the designer of the processor has to try to make programs that provoke erroneous behaviour. This was noticed when running randomized UVM tests (made by Mads Roligheten in another master thesis this year) that detected several bugs that was undetected after the standard asm tests we generated initially.

The simulation with test programs was done on both rtl and ntl.

7.7.1 Asynchronous challenges

More difficult to analyse at the simulation waveform, but no special challenges.

7.8 Physical implementation testing

For physical testing we have used a system as shown in figure 15. We have only used this system for fpga tests, but most of it will be reusable for asic tests.

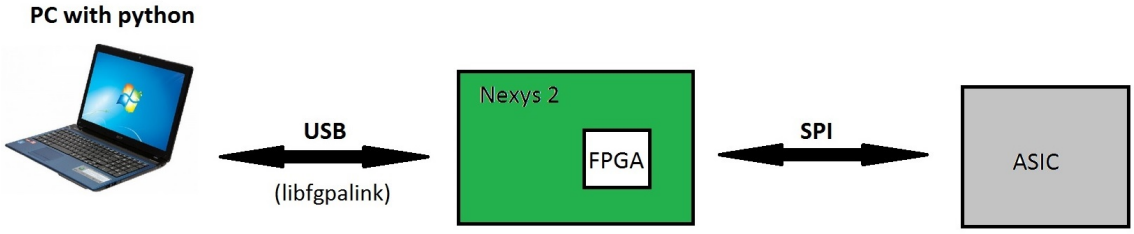


Figure 15: Test system

7.8.1 FPGA

The communication with the cpu is done through a bus interface. libfpgalink [9] is used for usb communication between the test computer and a 32 bit bus in the fpga. The cpu is has one address on the bus and the data packet controls the processor as described in figure 18 and table 2. The Digilent nexys 2 fpga board shown in figure 16 was used for testing. To verify the functionality, the registers are read out after execution of a program.

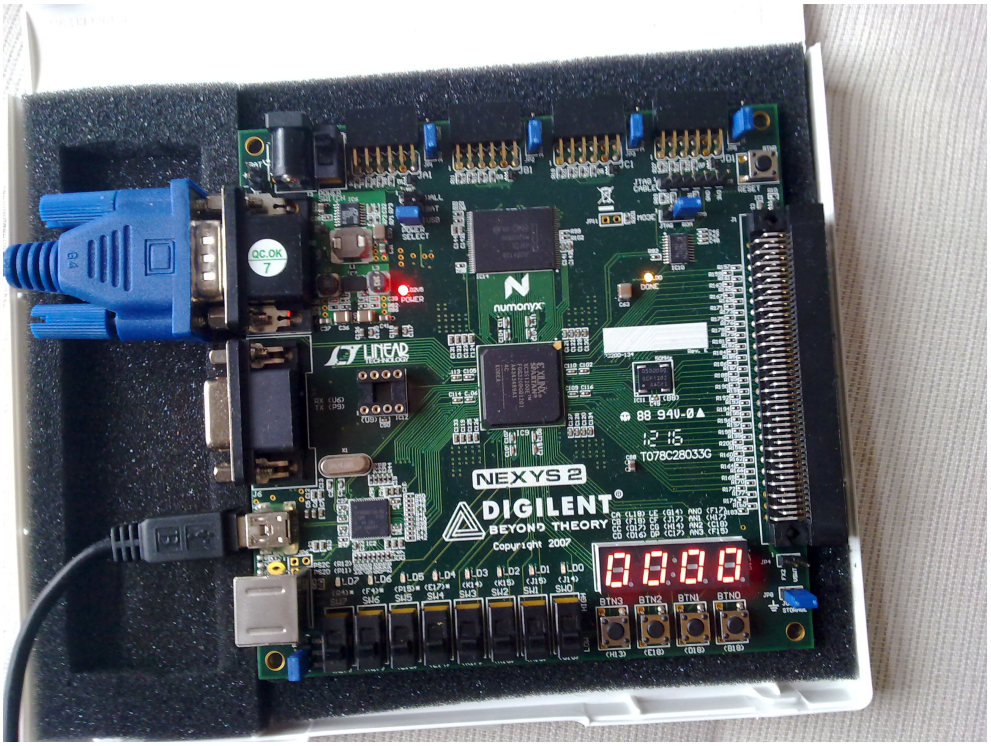


Figure 16: Digilent nexys 2 board

We also developed a vga interface to display data and settings in the CPU. This was used for the first tests, but discontinued as the usb interface allows a better 2 way communication (All test programs was uploaded to program memory at synthesis with the vga test bench. The PC-counter was incremented to start at the chosen program). Also the vga resolution was not compatible with newer screens and the vga hardware used a lot of space on the fpga. The usb interface also provided better runtime control, and adding additional functionality was faster and simpler.

Figure 17: VGA test bench GUI

Data bit	31 24	23 16	15 8	7 0
Function	Module	ID	VALUE	

Figure 18: USB/SPI data packet

Table 2: SPI/USB data control

Module	Function
0x0	Set clk dividor
0x1	Set delay on stage ID
0x2	Force CPU reset
0x3	Write Seld(control signal for clk muxes)
0x4	Write program line on address ID
0x5	Set register to read (VALUE)
0x6	Read the specified register
0x7	Turn off CPU clk (not yet implemented)
0x8	Send one CPU clock (not yet implemented)

7.8.2 ASIC

To test the asic the same system as the fpga tests would be used. The only difference is an Atmel developed semi spi link between the fpga board and the asic is added. The spi module inside the fpga and asic buffers up data packets and send them to the cpu.

This interface is untested (no ASIC has been produced), but 16 bit asynchronous SPI bus modules is ready for use. The cpu bus wrapper used for the fpga should be reusable with the spi module. The python scripts used on the computer side for libfpgalink communication should also be entirely reusable.

8 Results

8.1 Simulation

The first tests failed due to missing connections between request and acknowledge signals, but after a few modifications the simulations gave the expected results for all instructions for all processors. Note that not all branching instructions are fully implemented in the core.

8.2 Synthesis - ASIC

8.2.1 Area

For area measurements we only synthesize the cpu core with no memory, as this gives a better indication on how much the actual size changes. A successful synthesis was done by using the constraints given in appendix B.

- Synchronous - 16441
- Ringreq - Not synthesised
- Simple muller based dual clock - 21552
- Advanced muller FIFO - 31451

8.2.2 Performance

The performance is not quantified due to not having data about place and route timings available. Still, the performance is discussed in chapter 9 in an abstract way. The power after synthesis is reported to 50.9967 uW for the synchronous processor core without ram where 22 percent is the net switching power. The power for the simple muller asynchronous is reported to 55.9028 uW where 17 percent is the switching power. For the advanced version we have 1.5264 mW power where 14 percent is the net switching power. The power analysis was done with very basic constraints as this was not a priority, so they may be inaccurate.

8.3 Synthesis - FPGA

Only the asynchronous multi clock CPU and the synchronous CPU is synthesized for fpga. A successful synthesis was done by using the constraints given in appendix C.

8.4 Test programs - FPGA

Both processors worked with the test programs used in the simulations. You find all tests run in the attached .zip file in the flow/sim/tests folders. The asic test results was the same as for the simulated test programs.

8.5 Place and route

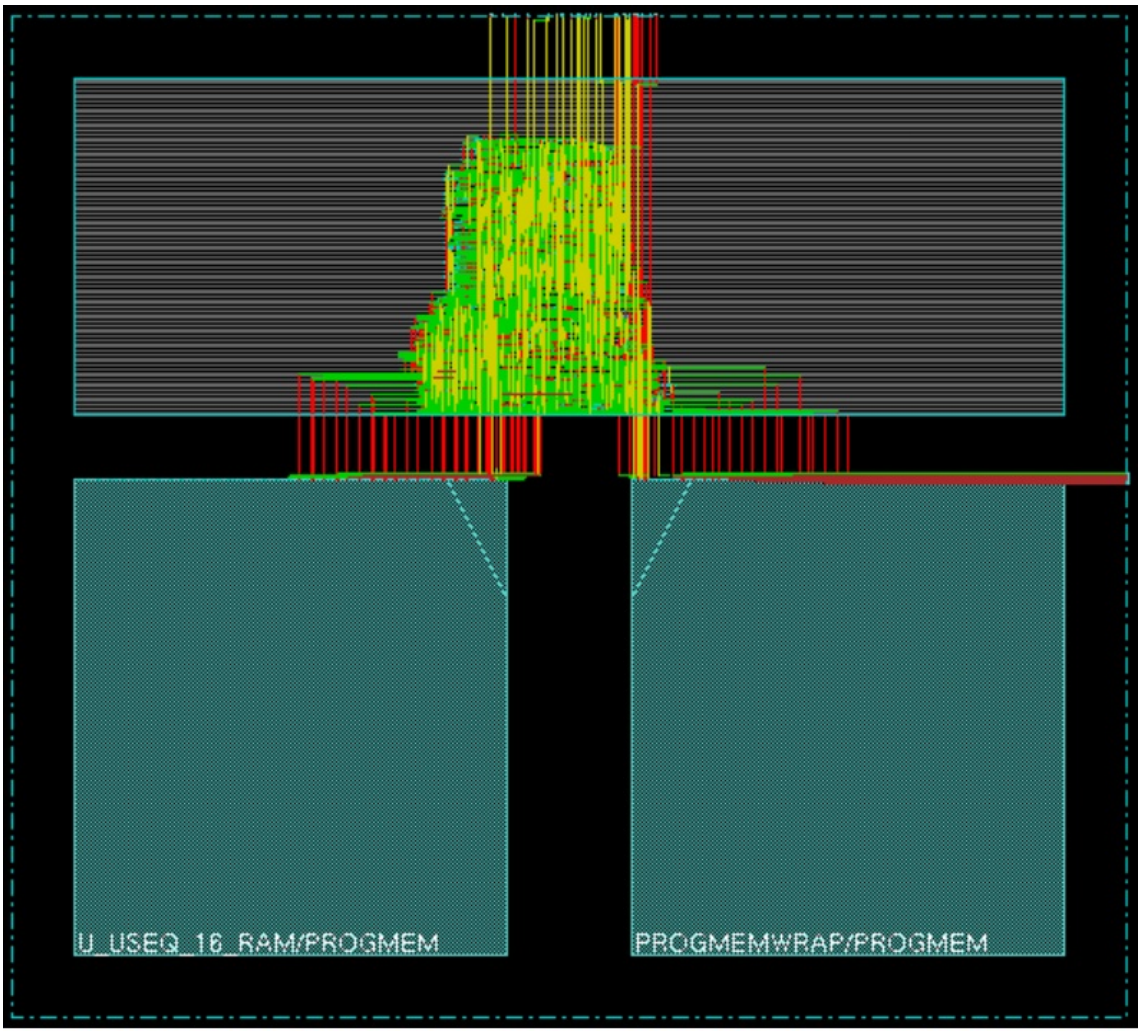


Figure 19: Early place and route of the synchronous processor

9 Discussion

9.1 Tools

The tools reacted as expected to the asynchronous design. Every module that exploited propagation delays and the modules dependent on modules that exploited was optimized away from the design when no constraint to keep them was added. The solution both for the asic synthesis and the fpga synthesis was to use constraints to keep the structure and nets that exploits propagation delays. We had to protect both the output inside the module and the output outside the module exploiting propagation delays to get a correct synthesis. The keep parameter worked a bit differently in Xilinx ISE and Design Compiler. In design compiler we had to protect the output of every instance of the modules that exploits propagation delays, but in Xlinx ISE we could protect the output of every instance of a module with one constraint, who makes the room for errors in Xilinx ISE smaller. Also the constraint file will be more compact and easier to keep updated when the design changes.

Simple asynchronous designs (like in this thesis) is straight forward to do with the existing tools. The only difference compared to synchronous design is the need for good constraints for the propagation delay modules and the need to verify that the synthesised design is as expected by reviewing the netlist schematics. This verification could be automated by running tests, but to pinpoint the problem and solve the errors, looking at the implementation of the modules is the simplest way to do it. For more complex designs, a look into assisting tools like petrify to design the asynchronous logic and more structured work by using for example petri nets, dependency graphs etc. would be beneficial. Also by using assisting tools and a more structured approach it would be possible for more persons to work on the same project without knowledge about the entire internal design of the processor.

9.2 Performance

All processors have very similar performance. This is due to no timing assumptions about the delay for each path in the cpu core is done during the design phase. For each extra latch and flip flop in the critical path of the design we have added a delay. Since our extra pipelining stages in the multi-stage advanced design does not give any advantages with this simple processor architecture, it is safe to say that the simple muller pipeline implementation is a better solution (less area, less power, higher speed) then the advanced multistaged solution with fifos.

When it comes to performance in terms of area,speed and power the results was as expected. The area increases with the complexity of the implementation. When the area increases, the power increases and the chip is slowed down (due to more elements in critical paths and unstrategical pipelining). Our asynchronous designs has larger delay lines then necessary and are designed for testing. This does increase the area as well.

9.3 Testability

The dual clock asynchronous design implemented on fpga and asic is good for running tests under different conditions. In order to do this, an actual asic is needed to extract precise and useful information. A work with different models for power, temperature and other conditions could be done so a designer could simulate the processor properly under a number of conditions.

9.4 Level of abstraction

When designing in verilog, a high level language is mostly preferred. For the propagation delay modules it was necessary to design on the gate level to ensure the correct functionality.

10 Conclusion

We successfully implemented asynchronous versions of our synchronous processor. The largest problem we experienced was when using propagation delays to generate or control signals. When exploiting propagation delays, you have to tell the synthesis tool that it should not optimize these modules away by giving it "keep" constraints. For the asic implementation with standard cell library it was not possible to protect whole modules because the synthesis tool would chose to use GTECH standard cells and not the ones in the Atmel standard cell library for generating the verilog netlist code. We worked around this problem by protecting only the input and the output wires from the modules. The "keep" constraints needed for a correct synthesis was the same for Xilinx ISE and Synopsys Design Compiler.

Correct reset of the circuit was another problem that showed up a few times. It is important that the whole pipeline is in a known state before processing starts.

When it comes to performance, our processor is operating very sequential and we had no gain in performance when implementing more complex control logic. However, we saw that also the more complex solutions synthesised correctly when adding the keep constraints to the propagation exploiting modules. We have also seen that it is possible to convert a synchronous processor just by changing the clock signals and adding delay chains and simple asynchronous controllers (very little changes are needed in the cpu core). It is also possible to do more advanced changes, but a good strategy is necessary to have a performance gain that could defend the larger area used that again affect the power.

Stability under variable conditions was an important target, but we have not been able to run tests considering the stability. This is a weakness of this thesis. Eighter good models for stability analysis of the place and route design or the actual asic is needed for testing. We have implemented programmable delays so it will be possible to test what delays should be used on the different stages under different conditions. Also by programming the delays it should be possible to run the processors under a lot of conditions. If implementing a dynamic delay scaling, it should be possible to have optimal performance under different conditions. It would be like dynamic clock scaling, but with the programmable delay elements you would not need to use the delay for the slowest stage, but could adapt for each programmable delay cell.

10.1 Future work

For this simple processor, the processing stage is very sequential (if we look away from the data memory load and store instructions). We will now introduce the changes needed in the design for different typical challenges that was not present in our processor.

If we start by looking at an interrupt, the processor would perform a jump to a given program address and then jump back when done. If the interrupt is dependent on the newest data exactly before the interrupt happens, one would have to be sure the newest values are available as an input to the ALU before running the instruction. This could be done by waiting the worst case for the data to propagate through the ALU, memory and register or the designer could reduce the penalty with register forwarding as done in our advanced multipipelined design. The problem with register forwarding is that every stage on the pipeline is asynchronous to each other so there is difficult to know if the data you read from another stage is in fact valid without using extensive timing assumptions that could be affected by the environment. However you know that the alu value from the instruction before is ready on the output of the alu and thus can be routed back to the input. In the multi staged implementation this problem was solved by having FIFOS that stored the last destinations to see if we had a dependency. If the dependency was in the last instruction, it would be possible to forward the result from the next stage, but if the dependency was several instructions before, we had to stall the processor until the data was available in the register. Here you could stall for a worst case time (using timing assumtions) or send dummy instructions on the pipeline to be sure that it is flushed (as we did in the advanced multipipelined design)

Both the FIFOS and registers for storing results gave a large area overhead, and more complex design. This problem has however little to do with the actual interrupt handling, but more on dependencies. To break out of the sequential operation of the processor when an interrupt is received you would only need to update the PC-counter to the correct address, store the current address and after the interrupt routine jump back. If there are no data dependencies you would not even have to care about the delay slots. Please note that you would probably need to store the registers to ram before running the isr (if you do not have interrupt registers) so this would create additional penalty.

If we were to add an extra peripheral, we can look on how we implemented the extra data memory. Simply said, we mux the req-ack to the correct module and use a delayline with the correct delay for that peripheral. This would increase the complexity for each element, but it would be a simple

upgrade.

When would you use assisting tools like petrify to generate control logic? In complex designs with many modules and buses and peripherals with a lot of pipeline stages and data dependencies. For five stages it is really controllable and not very difficult for our simple processor and the designer would spend a lot more time to learn new tools like this, however after using assisting tools one time and integrating them into a good flow, the risk of errors and the time needed to plan the upgrade would probably decrease. It would be interesting to compare the output from some tools to the solution chosen here.

Models and control logic for dynamic delay scaling would be interesting to look into.

Asic test with reduced power and different delay line settings.

Good models for generating timing assumptions to optimize designs is another thing to look into.

References and appendixes

References

[1] Jens Sparsø & Steve Furber, *Principles of Asynchronous Circuit Design - A Systems Perspective* Kluwer academic publishers, september 2001

[2] Ivan E. Sutherland, *Micropipelines* , ACM, june 1989

[3] *Aspida website* , http://www.ics.forth.gr/carv/async/activities_aspida.html

[4] Martin, Lines, Manohar, Nyström, Penzes, Southworth, Cummings, Lee, *The Design of an Asynchronous MIPS R3000 Microprocessor* , Department of ComputerScience California Instituteof Technology, unknown

[5] Furber, Garside, Riocreux, Temple, Day, Liu, Paver *Amulet2e: An Asynchronous Embedded Controller* , IEEE, 1999

[6] Woods, Day, Furber, Garside, Paver, Temple, *Amulet2e: An Asynchronous ARM Microprocessor* , IEEE, 1997

[7] Peter A. Beerel, Recep O. Ozdag, Marcos Ferretti, *A Designer's Guide to Asynchronous VLSI* , Cambridge University Press, 2010

[8] Ronan Barzic, *CTM16 instruction set*, Atmel, 2013

[9] Chris McClelland, *FPGALink User Manual Verilog Edition*, Makestuff, 2012

[10] Ronan Barzic, *Master theisis description*, Atmel, 2014

[11] *Archc project*, http://archc.sourceforge.net/, 2014

Appendix A - ZIP

Description about the files in the archive:

In the attached zip you will find the following files and software:

- asynchronous advanced - Advanced pipeline and simple pipeline core design
- asynchronous basic - Ring req ack designs
- asynchronous design - Design flow for asynchronous processor
- Fpgaproject - Design flow for fpga implementation:
- original design - Design flow folder for synchronous processor
- CTM16 instruction set.pdf - Instruction specification for the processor
- tpe ram 1536x8x4hs at110n n1.pdf - standard cell ram data sheet

A little elaboration about the design synchronous and asynchronous design flows: Most of the flow is removed when it comes to synthesis and the standard cells due to confidentiality. The standalone folder includes reports from synthesis and constraint files. Toolchain folder include the GNU compiler for ASM to program memory. Sim include the test bench and make files for simulation of rtl and ntl design. Note that the most designs uses standard cells not included (for example the ram and the delay cells. These can be replaced by the ram.v (simple ram) and a simple delay module with a simple delay instead of a chain of standard cell elements).

Appendix B - Constraint files ASIC

```
#System clk period
set clk_period 20

#Create system clk
create_clock -name clk -period $clk_period [get_ports
    clk]
#create_clock -name clk -period $clk_period [get_ports
    clk_apb_sysctrl]
#create_clock -name clk -period $clk_period [get_ports
    clk_test]
```

```
#create_clock -name clk -period $clk_period [get_ports
    osc_raw]

#Scan-Aware integrated clock gating cell
set clkgt_cell_dft gcksd3s8fhvt

#Set driving cell for the clocks, this helps CTS build
    a reasonable tree
set_driving_cell -lib_cell bufckd7s8fhvt [get_ports clk
    ]
#set_driving_cell -lib_cell bufckd7s8fhvt [get_ports
    clk_apb_sysctrl]
#set_driving_cell -lib_cell bufckd7s8fhvt [get_ports
    clk_test]
#set_driving_cell -lib_cell bufckd7s8fhvt [get_ports
    osc_raw]

if {$synopsys_program_name == "dc_shell"}  {

set_dont_touch ram_ack_muxed
set_dont_touch ram_ack_muxed

set_dont_touch ram_ack_muxed
set_dont_touch ram_req_stalled

set_dont_touch stall
set_dont_touch ram_clk
set_dont_touch ram_clk
set_dont_touch mem_clk
set_dont_touch clk
set_dont_touch clk
set_dont_touch mem_clk
set_dont_touch mem_clk_pre
set_dont_touch rf_clk
set_dont_touch pc_r_clk
set_dont_touch alu_req
set_dont_touch ram_req
set_dont_touch rf_done
set_dont_touch alu_ack
set_dont_touch mem_req_pre
set_dont_touch rf_req
```

```
set_dont_touch mem_req
set_dont_touch mem_clk_async
set_dont_touch dec_clk_async
set_dont_touch rf_clk_async
set_dont_touch ram_clk_async
set_dont_touch mem_req_delayed
set_dont_touch rf_req_pre
set_dont_touch PGRAMACK/out
set_dont_touch PGRAMDONE/out
set_dont_touch PGALUDONE/out
set_dont_touch PGRAM/out
set_dont_touch PGMEM/out
set_dont_touch PGRF/out
set_dont_touch PGPCR/out
set_dont_touch PGRAMACK
set_dont_touch PGRAMDONE
set_dont_touch PGALUDONE
set_dont_touch PGDEC/out
set_dont_touch U_RF/clk
set_dont_touch ALUD/DELMOD/in_temp
set_dont_touch ALUD/DELMOD/out
set_dont_touch RAMD/DELMOD/in_temp
set_dont_touch RAMD/DELMOD/out
set_dont_touch MEMD/DELMOD/in_temp
set_dont_touch MEMD/DELMOD/out
set_dont_touch RFD/DELMOD/in_temp
set_dont_touch RFD/DELMOD/out

set_dont_touch halt
set_dont_touch DECS/out
set_dont_touch SBALUAS0/out
set_dont_touch SBALUBS0/out
set_dont_touch SBOPMUXS0/out
set_dont_touch SBSELCS0/out
set_dont_touch SBRFWRS0/out
set_dont_touch SBWBMUXS0/out
set_dont_touch SBRFBS0/out
set_dont_touch SBPCMUXS0/out
set_dont_touch SBDECSTRS0/out
set_dont_touch SBWBDESTMUXS0/out
set_dont_touch SBDECRDS0/out
```

```
set_dont_touch SBSELCS2/out
set_dont_touch SBWBMUXS2/out
set_dont_touch SBRFWRS2/out
set_dont_touch SBALUCS2/out
set_dont_touch SBMEM8OS2/out
set_dont_touch SBMEM16OS2/out
set_dont_touch SBWBDESTMUXS2/out
set_dont_touch SBRFBS2/out
set_dont_touch SBDECRDS2/out
}


if {$synopsys_program_name == "pt_shell"} {
#Disable timing on SE
set_case_analysis 0 [get_ports {SCAN_ENABLE test_se SI
   }]
set_case_analysis 0 [get_ports {ramdsel aludsel rfdsel
   memdsel}]
set_case_analysis 1 [get_ports {sel_mode[0] sel_mode[1]
    sel_mode[2] sel_mode[3]}]
set_case_analysis 1 [get_ports {rst_n}]
#set_case_analysis 1 [get_nets {ram_req}]

set_disable_timing {RAMD ALUD MEMD RFD}
#group_path -name INOUT -from [get_nets {pc_r[1]}] -to
   [get_nets {pc_r[1]}]

}
```

Appendix C - Constraints FPGA

```
MODEL "simd_useq_async" keep_hierarchy = yes;
MODEL "simd_useq" keep_hierarchy = yes;
MODEL "progmem" keep_hierarchy = yes;
MODEL "fredivider" keep_hierarchy = yes;
MODEL "vga_draw" keep_hierarchy = yes;
MODEL "pulse_generator" keep_hierarchy = yes;
MODEL "delay_cell" keep_hierarchy = yes;
MODEL "delay_module" keep_hierarchy = yes;
MODEL "muller_c_element" keep_hierarchy = yes;
```

```
MODEL "ordelayelement" keep_hierarchy = yes;
MODEL "ordelayelement" keep_hierarchy = or;
BEGIN MODEL "delay_module"
        NET     "out" keep = yes;
        NET "in_temp" keep = yes;
        NET "zero" keep = yes;
END;

BEGIN MODEL "simd_useq_async"
        NET     "ram_clk" keep = yes;
        NET     "mem_clk" keep = yes;
        NET     "mem_clk_pre" keep = yes;
        NET     "rf_clk" keep = yes;
        NET     "pc_r_clk" keep = yes;
        NET     "alu_req" keep = yes;
        NET     "ram_req" keep = yes;
        NET     "rf_done" keep = yes;
        NET     "alu_ack" keep = yes;
        NET     "mem_req_pre" keep = yes;
        NET     "rf_req" keep = yes;
        NET     "mem_req" keep = yes;
        NET     "mem_clk_async" keep = yes;
        NET     "dec_clk_async" keep = yes;
        NET     "rf_clk_async" keep = yes;
        NET     "mem_req_delayed" keep = yes;
        NET     "rf_req_pre" keep = yes;
END;


BEGIN MODEL "ordelayelement"
        NET     "a" keep = yes;
        NET     "b" keep = yes;
        NET     "out" keep = yes;
END;
```

Appendix D - Project plan

Time in weeks	Event
2W	Litterature study
1W	Verilog study
1W	Atmel RTL flow study
1W	Write more verification programs
1W	Structural change
5W	Control logic
4W	RTL design/Verification/synthesis
1-3W	Physical implementation
3W	Report

Figure 20: Project plan

Appendix E - simple muller pipeline cpu implementation

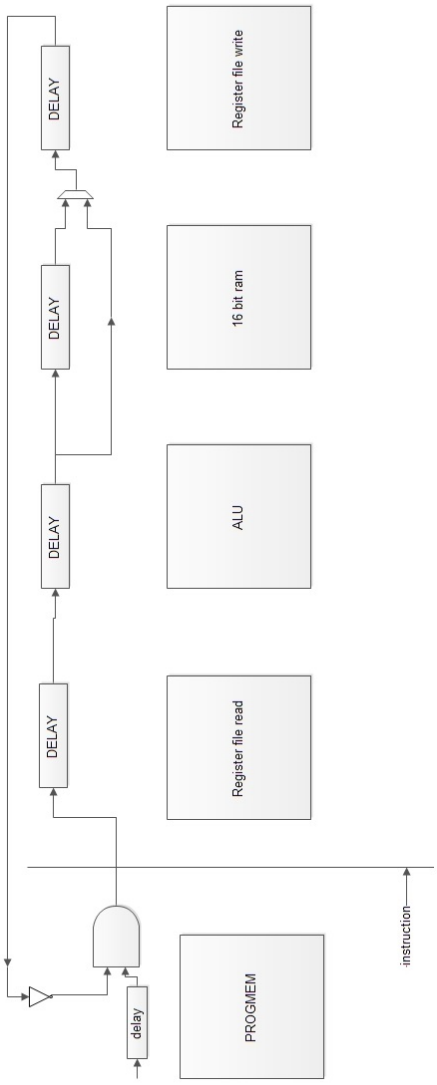


Figure 21: 2stageasync

Appendix F - CPU detailed schematics

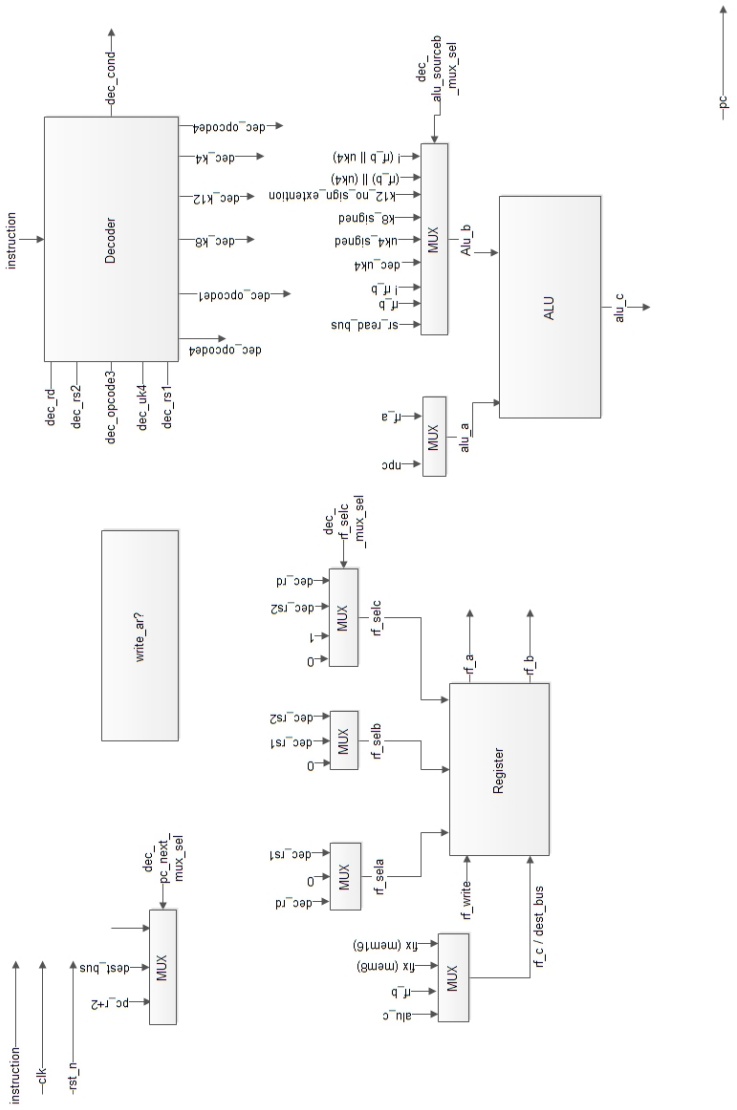


Figure 22: Detailed schematics

Appendix G - Multistage FIFO CPU detailed schematics

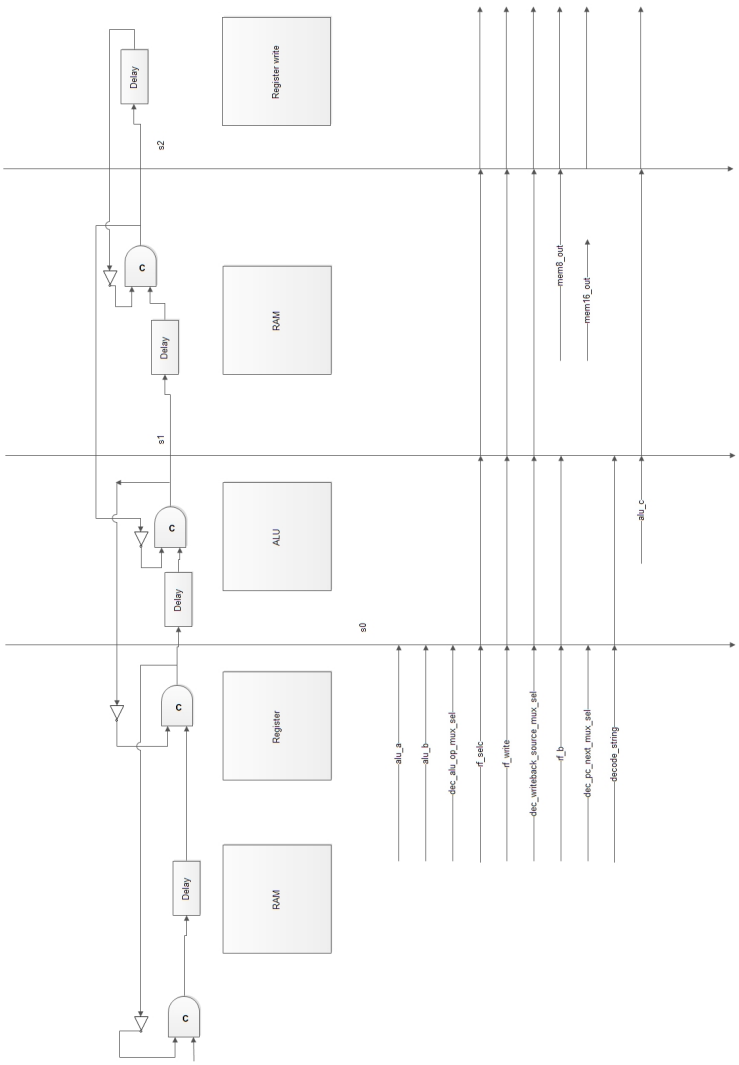


Figure 23: Schematic adv. async

Appendix H - Signal overview

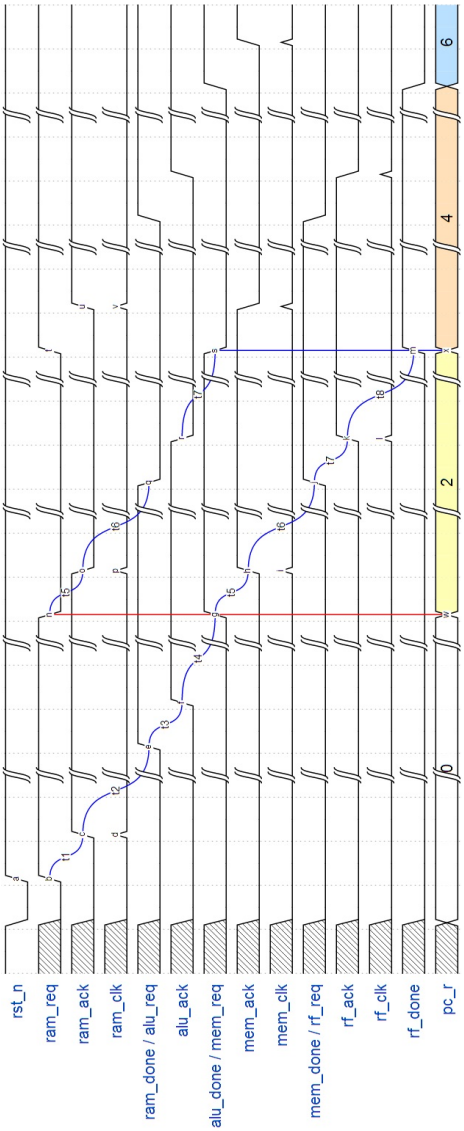


Figure 24: Async signal overview

Appendix I - Notes on other asynchronous processors

MIPS R3000

3 stage pipeline. Buffered cache. Asynchronous pipeline uses handshaking. Four phase dual rail.

Amulet1

32-bit RISC, ARM core, based on Sutherland's Micropipelines Each functional unit has an its own internal pipeline. Combination of 2 phase bundeled data based closeley on the Sutherland's micropipeline. But also conversion to 4 phase to control its level sensitive latches. (conventional transparent latch)

Amulet2e [5]

32- bit RISC. ARM core. Amulet1 was to demonstrate the possibility. It was broadly comparable to the synchronous version. Amulet2e is the enhanced version. 4 phase bundeed data. Not that deep pipeline (meaning fewer pipelines in each module). This was due to finding out that the depth of the pipeline on Amulet1 was to great. Note: This was due to FIFO buffers (conceptually easy to use within the micro pipeline design style), and too many where added.

[5, AMULET2e is a highly usable asynchronous embedded system chip. Its performance and power efficiency are competitive with the industry-leading clocked ARM designs, and in an idle loop its power reduces below that achievable in a clocked design without stopping the clock (whereafter the clocked chip takes considerable time to resume full performance). Its EMC properties are also very attractive, displaying a lower overall emission level and much less severe harmonic peaks than similar clocked circuits. Although in its current form AMULET2e has a minor logic fault and could not be tested economically in volume production, both of these issues could be remedied with a little design effort to deliver a highly effective, fully asynchronous controller for small embedded systems. AMULET2e incorporates several new architectural features and the means to evaluate them (the LRR, LLV, BTC, and abort handling can all be turned on or off independently). As such it contributes to the growing pool of architectural knowledge which must expand considerably further before asynchronous designers can compete with synchronous designers on equal terms. However, seeing is believing, and the reactions of the systems designers who have seen AMULET2e since prototypes first ran have been very favorable. Several prototype applications for the chip are presently under development. We sense that the barriers to the commercial exploitation of asynchronous design are beginning to fall.]

ASPIDA

Open source IP for DLX instruction set, RISC architecture 32 bit 5 stage pipeline RISC. No more information found in the quick research then that

the clock is replaced by handshaking controllers. Flip-flops are replaced by pair latches.

Appendix J - Modifications and fixed to the synchronous processor

In this section some bugs we noticed in the initial synchronous design are listed.

J.A - Multiple drivers

NPC and aluA (2 cases with same condition).Fixed.

J.B - Mux for selb did not include dedRD as input

This caused failure of addressing of the extra memory. Fixed.

J.C - LDIR1 command sets MUX selb to RS2

Can cause propagating X during simulation due to register file is only 15 rows long. Selb should be set to R0 for this instruction and the register should be 16 rows long. Unfixed.

J.D - Branch not implemented properly - added for instr. ADD and cond equal in RTL for synchronous version

Branch is not prioritized, and not fully implemented. Not fixed.

J.E - -pc r counts 2

This is done purposely. PC is shifted one to the right and used as address for the memory.

J.F - Carry bits on alu wrong direction

Simple to fix, but not fixed. (Fix is dependent on what we want to use it for)

J.G - Adding extra memory

The processor had instructions implemented for loading word, byte and storing word and byte. However, the ram interface was not implemented in the processor. This was added. Both negative edge triggered and stalling on loading. The stall is implemented by muxing away the register file clock and running the same instruction one more time. Since we used equal ram blocks for program memory and memory, the stalling is the best solution as the ram is probably the slowest module on the chip. And by adding 1 ram read out on negative edge, we would have to double the clock period on the chip.

J.H - Removing delay slots /pc-r stage

The CPU originally had registers both for pc_r and npc (next progam counter). The timing is shown in figure 25. This caused the CPU two have two delay slots. By removing the register for the npc we only have to 1 delay slot left (shown in figure 26).

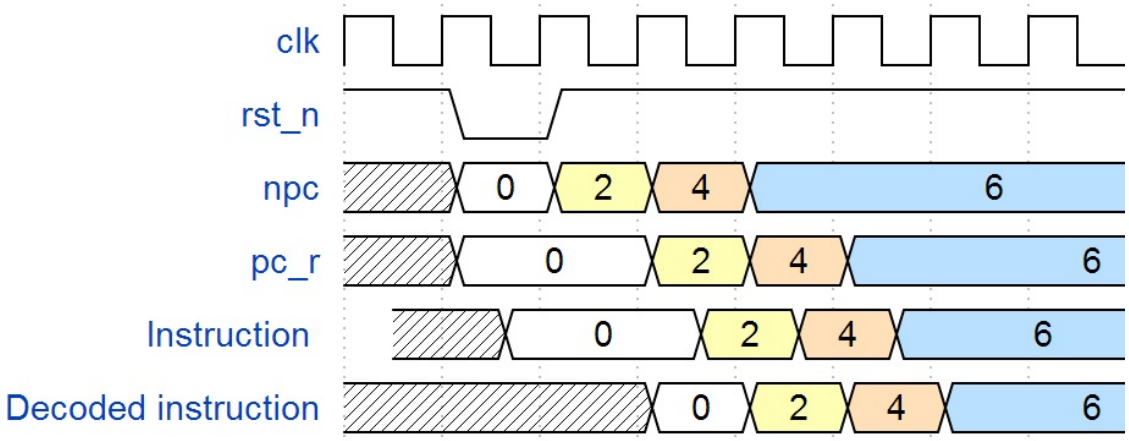


Figure 25: npc pc r delay slot

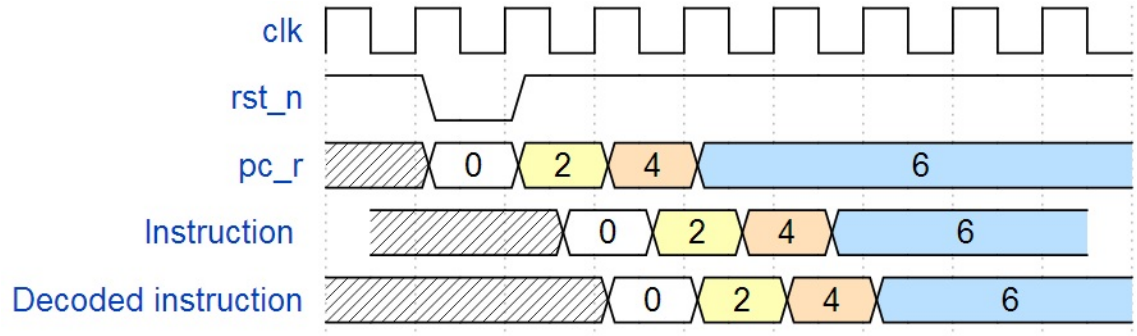


Figure 26: pc r delay slot

Appendix K - Verilog sources

We recommend to see the attached zip file for all sources, but since the report should be self contained, the important files are included here.

K.A - Parameters

```
/* Parameter file for SIMD module */

// Should be moved in a feature file ?
parameter USEQ_INST_BITS = 16,
  USEQ_INST_MSB = USEQ_INST_BITS-1;

parameter USEQ_WORD_BITS = 16,
  USEQ_WORD_MSB = USEQ_WORD_BITS-1;

  parameter USEQ_BYTE_BITS = 16,
  USEQ_BYTE_MSB = USEQ_BYTE_BITS-1;

parameter USEQ_HALF_WORD_BITS = USEQ_WORD_BITS/2;

parameter USEQ_ADDR_BITS = USEQ_WORD_BITS;

parameter USEQ_WORD_MINUS_K4 = USEQ_WORD_BITS-4;
parameter USEQ_WORD_MINUS_K8 = USEQ_WORD_BITS-8;
parameter USEQ_WORD_MINUS_K12 = USEQ_WORD_BITS-12;

// default increment for the PC
```

```
parameter USEQ_PC_INC = 2;

// Regfile
parameter USEQ_RF_SEL_BITS = 4,
  USEQ_RF_SEL_MSB = USEQ_RF_SEL_BITS-1;



//@begin[inst_format]
parameter USEQ_INST_FORMAT_OPCODE1_SIZE = 4;
parameter USEQ_INST_FORMAT_OPCODE1_MSB = 3;
parameter USEQ_INST_FORMAT_OPCODE1_OFFSET = 12;

parameter USEQ_INST_FORMAT_RD_SIZE = 4;
parameter USEQ_INST_FORMAT_RD_MSB = 3;
parameter USEQ_INST_FORMAT_RD_OFFSET = 8;

parameter USEQ_INST_FORMAT_OPCODE3_SIZE = 4;
parameter USEQ_INST_FORMAT_OPCODE3_MSB = 3;
parameter USEQ_INST_FORMAT_OPCODE3_OFFSET = 4;

parameter USEQ_INST_FORMAT_RS2_SIZE = 4;
parameter USEQ_INST_FORMAT_RS2_MSB = 3;
parameter USEQ_INST_FORMAT_RS2_OFFSET = 0;

parameter USEQ_INST_FORMAT_UK4_SIZE = 4;
parameter USEQ_INST_FORMAT_UK4_MSB = 3;
parameter USEQ_INST_FORMAT_UK4_OFFSET = 0;

parameter USEQ_INST_FORMAT_RS1_SIZE = 4;
parameter USEQ_INST_FORMAT_RS1_MSB = 3;
parameter USEQ_INST_FORMAT_RS1_OFFSET = 4;

parameter USEQ_INST_FORMAT_OPCODE4_SIZE = 4;
parameter USEQ_INST_FORMAT_OPCODE4_MSB = 3;
parameter USEQ_INST_FORMAT_OPCODE4_OFFSET = 0;

parameter USEQ_INST_FORMAT_K8_SIZE = 8;
parameter USEQ_INST_FORMAT_K8_MSB = 7;
parameter USEQ_INST_FORMAT_K8_OFFSET = 0;
```

```
parameter USEQ_INST_FORMAT_COND_SIZE = 4;
parameter USEQ_INST_FORMAT_COND_MSB = 3;
parameter USEQ_INST_FORMAT_COND_OFFSET = 8;

parameter USEQ_INST_FORMAT_K12_SIZE = 12;
parameter USEQ_INST_FORMAT_K12_MSB = 11;
parameter USEQ_INST_FORMAT_K12_OFFSET = 0;

parameter USEQ_INST_FORMAT_K4_SIZE = 4;
parameter USEQ_INST_FORMAT_K4_MSB = 3;
parameter USEQ_INST_FORMAT_K4_OFFSET = 0;

//@end[inst_format]



//@begin[param_opcode]
// Opcodes used for decoding : ['opcode1', 'opcode3', '
   opcode4', 'cond']

parameter USEQ_OPCODES_AND = 'b01110000????????;
parameter USEQ_OPCODES_XNORR1 = 'b10000011????????;
parameter USEQ_OPCODES_ADD2C = 'b01111000????????;
parameter USEQ_OPCODES_LDIR1 = 'b1111????????????;
parameter USEQ_OPCODES_ORR1 = 'b10000001????????;
parameter USEQ_OPCODES_ANDR1 = 'b10000000????????;
parameter USEQ_OPCODES_XORR1 = 'b10000010????????;
parameter USEQ_OPCODES_LB = 'b0100????????????;
parameter USEQ_OPCODES_SUB2C = 'b01111001????????;
parameter USEQ_OPCODES_RET = 'b1110????0010????;
parameter USEQ_OPCODES_LW = 'b0011????????????;
parameter USEQ_OPCODES_SUB3 = 'b1101????????????;
parameter USEQ_OPCODES_XNOR = 'b01110011????????;
parameter USEQ_OPCODES_BRPC = 'b1001????????????;
parameter USEQ_OPCODES_SUB2 = 'b0010????????????;
parameter USEQ_OPCODES_HALT = 'b1110????0011????;
parameter USEQ_OPCODES_MFSR = 'b1110????0001????;
parameter USEQ_OPCODES_MTSR = 'b1110????0000????;
parameter USEQ_OPCODES_ORI4 = 'b0000????????????;
parameter USEQ_OPCODES_SRAI = 'b10000101????????;
parameter USEQ_OPCODES_SEXT = 'b01110111????????;
```

```
parameter USEQ_OPCODES_BR = 'b1010???????????;
parameter USEQ_OPCODES_XOR = 'b01110010????????;
parameter USEQ_OPCODES_SRA = 'b01110100????????;
parameter USEQ_OPCODES_ADD3 = 'b1100???????????;
parameter USEQ_OPCODES_ADD2 = 'b0001???????????;
parameter USEQ_OPCODES_SRLI = 'b10000110????????;
parameter USEQ_OPCODES_SW = 'b0101???????????;
parameter USEQ_OPCODES_SRL = 'b01110101????????;
parameter USEQ_OPCODES_SLL = 'b01110110????????;
parameter USEQ_OPCODES_SLLI = 'b10000111????????;
parameter USEQ_OPCODES_SB = 'b0110???????????;
parameter USEQ_OPCODES_OR = 'b01110001????????;
//@end[param_opcode]

//@begin[rf_sela]
parameter USEQ_SEL_RFA_RD = 0;
parameter USEQ_SEL_RFA_R0 = 1;
parameter USEQ_SEL_RFA_RS1 = 2;
parameter USEQ_SEL_RFA_BITS = 2;
parameter USEQ_SEL_RFA_MSB = 1;
//@end[rf_sela]

//@begin[rf_selb]
parameter USEQ_SEL_RFB_RD = 0;
parameter USEQ_SEL_RFB_R14 = 1;
parameter USEQ_SEL_RFB_R1 = 2;
parameter USEQ_SEL_RFB_RS2 = 3;
parameter USEQ_SEL_RFB_RS1 = 4;
parameter USEQ_SEL_RFB_BITS = 3;
parameter USEQ_SEL_RFB_MSB = 2;
//@end[rf_selb]

//@begin[rf_selc]
parameter USEQ_SEL_RFC_RD = 0;
parameter USEQ_SEL_RFC_R0 = 1;
parameter USEQ_SEL_RFC_R1 = 2;
parameter USEQ_SEL_RFC_RS2 = 3;
parameter USEQ_SEL_RFC_BITS = 2;
parameter USEQ_SEL_RFC_MSB = 1;
//@end[rf_selc]
```

```
//@begin[alu_sourcea]
parameter USEQ_SEL_ALUA_RFA = 0;
parameter USEQ_SEL_ALUA_NPC = 1;
parameter USEQ_SEL_ALUA_BITS = 1;
parameter USEQ_SEL_ALUA_MSB = 0;
//@end[alu_sourcea]

//@begin[alu_sourceb]
parameter USEQ_SEL_ALUB_K12 = 0;
parameter USEQ_SEL_ALUB_RFB_OR_UK4 = 1;
parameter USEQ_SEL_ALUB_SR = 2;
parameter USEQ_SEL_ALUB_UK4 = 3;
parameter USEQ_SEL_ALUB_NOT_RFB = 4;
parameter USEQ_SEL_ALUB_RFB = 5;
parameter USEQ_SEL_ALUB_K4 = 6;
parameter USEQ_SEL_ALUB_K8 = 7;
parameter USEQ_SEL_ALUB_NOT_RFB_OR_UK4 = 8;
parameter USEQ_SEL_ALUB_BITS = 4;
parameter USEQ_SEL_ALUB_MSB = 3;
//@end[alu_sourceb]

//@begin[wb_source]
parameter USEQ_SEL_WBSOURCE_MEM16 = 0;
parameter USEQ_SEL_WBSOURCE_RFB = 1;
parameter USEQ_SEL_WBSOURCE_ALU = 2;
parameter USEQ_SEL_WBSOURCE_MEM8 = 3;
parameter USEQ_SEL_WBSOURCE_BITS = 2;
parameter USEQ_SEL_WBSOURCE_MSB = 1;
//@end[wb_source]

//@begin[wb_dest]
parameter USEQ_SEL_DEST_MEM16 = 0;
parameter USEQ_SEL_DEST_SR = 1;
parameter USEQ_SEL_DEST_COND_LINK_REG = 2;
parameter USEQ_SEL_DEST_RFC = 3;
parameter USEQ_SEL_DEST_PC = 4;
parameter USEQ_SEL_DEST_MEM8 = 5;
parameter USEQ_SEL_DEST_BITS = 3;
parameter USEQ_SEL_DEST_MSB = 2;
//@end[wb_dest]
```

```
//@begin[alu_op]
parameter USEQ_SEL_ALUOP_AND = 0;
parameter USEQ_SEL_ALUOP_XNOR = 1;
parameter USEQ_SEL_ALUOP_SRA = 2;
parameter USEQ_SEL_ALUOP_XOR = 3;
parameter USEQ_SEL_ALUOP_SUB = 4;
parameter USEQ_SEL_ALUOP_SRL = 5;
parameter USEQ_SEL_ALUOP_SLL = 6;
parameter USEQ_SEL_ALUOP_ADD = 7;
parameter USEQ_SEL_ALUOP_SEXT = 8;
parameter USEQ_SEL_ALUOP_OR = 9;
parameter USEQ_SEL_ALUOP_BITS = 4;
parameter USEQ_SEL_ALUOP_MSB = 3;
//@end[alu_op]

//@begin[pc_udate]
parameter USEQ_SEL_PC_COND_LOAD_FROM_ALU = 0;
parameter USEQ_SEL_PC_FROM_ALU = 1;
parameter USEQ_SEL_PC_INCREMENT = 2;
parameter USEQ_SEL_PC_BITS = 2;
parameter USEQ_SEL_PC_MSB = 1;
//@end[pc_update]

//@begin[param_cond]
parameter USEQ_COND_GT = 4'd4;
parameter USEQ_COND_GEU = 4'd9;
parameter USEQ_COND_ALAL = 4'd15;
parameter USEQ_COND_AL = 4'd14;
parameter USEQ_COND_GE = 4'd5;
parameter USEQ_COND_LEU = 4'd7;
parameter USEQ_COND_EQ = 4'd0;
parameter USEQ_COND_C = 4'd10;
parameter USEQ_COND_LE = 4'd3;
parameter USEQ_COND_LTU = 4'd6;
parameter USEQ_COND_O = 4'd11;
parameter USEQ_COND_LT = 4'd2;
parameter USEQ_COND_GTU = 4'd8;
parameter USEQ_COND_OU = 4'd0;
parameter USEQ_COND_NEQ = 4'd1;
```

```
//@end[param_cond]

//@begin[param_sreg]
parameter USEQ_SR_AR0 = 4'd8;
parameter USEQ_SR_SR = 4'd0;
parameter USEQ_SR_AR2 = 4'd10;
parameter USEQ_SR_AR1 = 4'd9;

//@end[param_sreg]

//@begin[flow]
parameter USEQ_FLOW_HALT_YES = 0;
parameter USEQ_FLOW_HALT_NO = 1;
parameter USEQ_FLOW_HALT_BITS = 1;
parameter USEQ_FLOW_HALT_MSB = 0;
//@end[flow]
```

K.B - Register file

```
module simd_useq_rf (/*AUTOARG*/
  // Outputs
  rf_a, rf_b,
  // Inputs
  rf_sela, rf_selb, rf_selc, rf_c, rf_write, clk, rst_n
  );

  parameter NUM_REGS=15;
`include "simd_params.v"

  // Selection
  input  [USEQ_RF_SEL_MSB:0] rf_sela;
  input [USEQ_RF_SEL_MSB:0]  rf_selb;
  input [USEQ_RF_SEL_MSB:0]   rf_selc;
  // Data ports
  output [USEQ_WORD_MSB:0] rf_a;
  output [USEQ_WORD_MSB:0] rf_b;
  input [USEQ_WORD_MSB:0]  rf_c;
  input                    rf_write;
  input                    clk;
  input                    rst_n;
```

```
/*AUTOINPUT*/

/*AUTOOUTPUT*/

/*AUTOREG*/
// Beginning of automatic regs (for this module's
   undeclared outputs)
reg [USEQ_WORD_MSB:0] rf_a;
reg [USEQ_WORD_MSB:0] rf_b;
// End of automatics

/*AUTOWIRE*/




reg [USEQ_WORD_MSB:0] regfile [1:NUM_REGS-1];

always @(*) begin
  if(rf_sela != 0) begin
    rf_a <= regfile[rf_sela];
  end
  else begin
    rf_a <= 0;
  end
end


always @(*) begin
  if(rf_selb != 0) begin
    rf_b <= regfile[rf_selb];
  end
  else begin
    rf_b <= 0;
  end
end

integer k;

always @(posedge clk or negedge rst_n) begin
      if(rst_n == 0) begin
```

```
                for (k = 1; k < NUM_REGS; k = k + 1)
                    begin
                        regfile[k] <= 0;
                    end
    end else if((rf_selc != 0) && rf_write) begin
      regfile[rf_selc] <= rf_c;
    end
  end


  // For debugging

// synthesis translate_off


wire   [31:0] r0 ;
wire   [31:0] r1 ;
wire   [31:0] r2 ;
wire   [31:0] r3 ;
wire   [31:0] r4 ;
wire   [31:0] r5 ;
wire   [31:0] r6 ;
wire   [31:0] r7 ;
wire   [31:0] r8 ;
wire   [31:0] r9 ;
wire   [31:0] r10;
wire   [31:0] r11;
wire   [31:0] r12;
wire   [31:0] r13;
wire   [31:0] r14;




assign      r1 =        regfile[1];
assign      r2 =        regfile[2];
assign      r3 =        regfile[3];
assign      r4 =        regfile[4];
assign      r5 =        regfile[5];
```

```
assign      r6 =          regfile[6];
assign      r7 =          regfile[7];
assign      r8 =          regfile[8];
assign      r9 =          regfile[9];
assign      r10 =         regfile[10];
assign      r11 =         regfile[11];
assign      r12 =         regfile[12];
assign      r13 =         regfile[13];
assign      r14 =         regfile[14];



// synthesis translate_on




endmodule // dlx3_regfile
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.C - Decoder

```
module simd_useq_decoder(dec_opcode1,dec_rd,dec_opcode3
    ,dec_rs2,dec_uk4,dec_rs1,dec_opcode4,dec_k8,dec_cond
    ,dec_k12,dec_k4,instruction,clk,rst_n);
`include "simd_params.v"
input [USEQ_INST_MSB:0] instruction;
input clk;
input rst_n;
output [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1;
output [USEQ_INST_FORMAT_RD_MSB:0] dec_rd;
output [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3;
output [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2;
output [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4;
```

```
output [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1;
output [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4;
output [USEQ_INST_FORMAT_K8_MSB:0] dec_k8;
output [USEQ_INST_FORMAT_COND_MSB:0] dec_cond;
output [USEQ_INST_FORMAT_K12_MSB:0] dec_k12;
output [USEQ_INST_FORMAT_K4_MSB:0] dec_k4;



  reg [USEQ_INST_MSB:0]      dec_inst_r;


wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE1_OFFSET +:
   USEQ_INST_FORMAT_OPCODE1_SIZE];

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd  = dec_inst_r[
   USEQ_INST_FORMAT_RD_OFFSET +:
   USEQ_INST_FORMAT_RD_SIZE];

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE3_OFFSET +:
   USEQ_INST_FORMAT_OPCODE3_SIZE];
//wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3  =
   dec_inst_r[15:12];

wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2  = dec_inst_r
   [USEQ_INST_FORMAT_RS2_OFFSET +:
   USEQ_INST_FORMAT_RS2_SIZE];

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4  = dec_inst_r
   [USEQ_INST_FORMAT_UK4_OFFSET +:
   USEQ_INST_FORMAT_UK4_SIZE];

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1  = dec_inst_r
   [USEQ_INST_FORMAT_RS1_OFFSET +:
   USEQ_INST_FORMAT_RS1_SIZE];

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE4_OFFSET +:
   USEQ_INST_FORMAT_OPCODE4_SIZE];
```

```
wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8  = dec_inst_r[
   USEQ_INST_FORMAT_K8_OFFSET +:
   USEQ_INST_FORMAT_K8_SIZE];

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond  =
   dec_inst_r[USEQ_INST_FORMAT_COND_OFFSET +:
   USEQ_INST_FORMAT_COND_SIZE];

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12  = dec_inst_r
   [USEQ_INST_FORMAT_K12_OFFSET +:
   USEQ_INST_FORMAT_K12_SIZE];

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4  = dec_inst_r[
   USEQ_INST_FORMAT_K4_OFFSET +:
   USEQ_INST_FORMAT_K4_SIZE];


  always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      /*AUTORESET*/
      // Beginning of autoreset for uninitialized flops
      dec_inst_r <= {(1+(USEQ_INST_MSB)){1'b0}};
      // End of automatics
    end
    else begin
      dec_inst_r <= instruction;

    end
  end




endmodule
```

K.D - Pulse generator

```
`timescale 1ns / 1ps
```

```
module pulse_generator(out,in,sel);
        input in;
        output out;
        wire in;
        wire out;
        wire a;
        input [3:0] sel;
        delay_cell  #(.DELAY (16)) D1(.z(a),.i(in),.sel
           (sel));
        xor PXOR(out, a, in);
endmodule
```

K.E - Delay elements

delaycell

```
`timescale 1ns / 1ps
module delay_cell (z,i,sel);

        function integer clog2;
                input integer value;
                begin
                        value = value-1;
                        for (clog2=0; value>0; clog2=
                           clog2+1)
                                value = value>>1;
                end
        endfunction

        parameter DELAY = 1;
        parameter NR_BITS = clog2(DELAY) -1;
        input i;
        input [3:0] sel;
        output z;
        wire [NR_BITS:0] delay_sel;
        assign delay_sel = sel[3:0];
        delay_module #(.DELAY(DELAY)) DELMOD(.out(z),.
           in(i),.sel(delay_sel[NR_BITS:0]));
        //assign #DELAY z = i;
endmodule
```

delaymodule

```
`timescale 1ns / 1ps
```

```
module delay_module(out,in,sel) ;
        function integer clog2;
                input integer value;
                begin
                        value = value-1;
                        for (clog2=0; value>0; clog2=
                            clog2+1)
                                value = value>>1;
                end
        endfunction


        parameter DELAY = 4;
        parameter NR_BITS = clog2(DELAY) -1 ;
        input in;
        input [NR_BITS:0] sel;
        output out;
        reg out;
        //ORDLY15s8fhvt DELAYEL(out,in,1'b1);
        wire [DELAY-1:0] in_temp;
        wire test;
        wire zero;
        assign zero = 1'b0;
        genvar i;
                ordelayelement DELAYEL(.out(in_temp[0])
                    ,.a(in),.b(zero));
        generate
                for(i=0;i<DELAY-1; i=i+1) begin:
                    delaygen
                                ordelayelement DELAYEL
                                    (.out(in_temp[i+1])
                                    ,.a(in_temp[i]),.b(
                                    zero));
                end
        endgenerate

        always @ * begin
                out     <= in_temp[sel];
        end
```

```
endmodule
```

K.F - simple RAM

```
// Generic RAM model
`timescale 1ns/1ps


module ram (/*AUTOARG*/
   // Outputs
   read_data,
   // Inputs
   clock, addr, write_data, cs, we
   );
   parameter WORD_WIDTH = 16;

   parameter ADDR_WIDTH = 8;
   parameter RAM_SIZE = 1<< ADDR_WIDTH;

   localparam WORD_MSB = WORD_WIDTH-1;


   localparam ADDR_MSB = ADDR_WIDTH-1;

   input clock;
   input [ADDR_MSB:0] addr;
   input [WORD_MSB:0] write_data;

   input              cs;
   input              we;
   output [WORD_MSB:0] read_data;

   /*AUTOINPUT*/

   /*AUTOOUTPUT*/

   /*AUTOREG*/
   // Beginning of automatic regs (for this module's
      undeclared outputs)
   reg [WORD_MSB:0]       read_data;
```

```
  // End of automatics

  /*AUTOWIRE*/


  reg [WORD_MSB:0]       mem0 [0:RAM_SIZE-1];

  reg [ADDR_MSB:0] addr_latched;


  always @(posedge clock) begin
    if(cs) begin
      addr_latched <= addr;
    end
  end


  always @(posedge clock) begin
    #3;
    if(cs)
      read_data <= mem0[addr_latched[ADDR_MSB:0]];
  end



  // Write

 always @(posedge clock) begin
    if(cs && we) begin
      $display("Memory write ");
      mem0[addr] <= write_data;
    end
 end

  integer i;

//  initial begin
//    #10;
//
//    for(i=0;i<10;i=i+1) begin
//      $display("%d : %x",i,mem0[i]);
```

```
//
//      end
//   end
//

endmodule // ram
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.G - Memwrapper

```
module mem_wrapper(
        read_data,
        clk,
    addr,
    write_data,
    cs,
    we);


input [15:0] addr;
input clk;
input cs;
input we;
input [15:0] write_data;
output [15:0] read_data;

wire [31:0] read_data_temp;
reg [10:0] adr;
reg [3:0] we_in;
reg [3:0] re;

assign read_data = read_data_temp[15:0];

always @* begin
if(we == 1'b1) begin
```

```
                                    we_in = 4'b0011;
                                    re = 4'b0000;
        end else begin
                                    we_in = 4'b0000;
                            re = 4'b0011;
end
end



tpe_ram_1536x8x4hs_at110n_n1 PROGMEM (
        .clk    (clk),
        .me             (cs),
        .re             (re[3:0]),
        .we             (we_in[3:0]),
        .adr    (addr[10:0]),
        .din    ( {16'b0, write_data}),// |
           instruction_in[USEQ_INST_MSB:0]),
        .rm             (4'b1111),
        .dout   (read_data_temp[31:0]),
        .pd             (1'b0),
        .pdd    (1'b1));

endmodule
```

K.H - progmemwrapper

```
module prog_mem_wrapper(
        load_program,
        prog_clk,
        prog_adr,
    clk,
    pc_r,
    instruction_in,
    instruction,
        rst_n_in,
        rst_n_out);

input load_program;
input prog_clk;
input [10:0] prog_adr;
```

```
input    clk;
input    [15:0] pc_r;
input    [15:0] instruction_in;
input rst_n_in;
output rst_n_out;
output  [15:0] instruction;


reg rst_n_out;
wire [31:0] instruction_temp;
assign instruction = instruction_temp[15:0];
reg ram_clk;
reg [10:0] adr;
reg [3:0] we;
reg [3:0] re;

always @* begin
        if(load_program ==1) begin
                rst_n_out = 1'b0;
        end else begin
                rst_n_out = rst_n_in;
        end
end

always @* begin
if(load_program == 1'b1) begin
        adr = prog_adr[10:0];
                                we = 4'b0011;
                        re = 4'b0000;
        end else begin
                                we = 4'b0000;
                        re = 4'b0011;
        adr = pc_r[11:1];
end
end

        always @* begin
                if(load_program == 1'b1) begin
                        ram_clk = #10 prog_clk;

                end else begin
```

```
                        ram_clk = #10 clk;

                end
        end

tpe_ram_1536x8x4hs_at110n_n1 PROGMEM (
        .clk    (ram_clk),
        .me             (1'b1),
        .re             (re[3:0]),
        .we             (we[3:0]),
        .adr    (adr[10:0]),
        .din    ( {16'b0, instruction_in}),// |
          instruction_in[USEQ_INST_MSB:0]),
        .rm             (4'b1111),
        .dout   (instruction_temp[31:0]),
        .pd             (1'b0),
        .pdd    (1'b1));

endmodule
```

K.I - sync chip top

```
`timescale 1ns/1ps
//`include "simd_params.v"

module chip_top (/*AUTOARG*/
  // Outputs
  cpu_ack,
  // Inputs
  sel_mode ,rst_n_temp, cpu_req, load_program, prog_clk
     , clk,ramdsel,aludsel,rfdsel,memdsel,
     instruction_in,prog_adr,pc_r
  );

  input rst_n_temp;                     // To U_USEQ of
     simd_useq.v
  input load_program;
  input prog_clk;
  input clk;
  input [3:0] ramdsel;
  input [3:0] aludsel;
```

```
input [3:0] rfdsel;
      input [3:0] memdsel;

// End of automatics
/*AUTOOUTPUT*/
// Beginning of automatic outputs (from unused
   autoinst outputs)
output [15:0] pc_r;                    // From U_USEQ of
   simd_useq.v
output cpu_ack;
input  cpu_req;
input [3:0] sel_mode;
// End of automatics
wire mem_cs,mem_we;
wire  [15:0] mem_addr;
wire [15:0] mem16_out;
wire [15:0] mem_in;
  wire [15:0] mem_out;
      input [10:0]prog_adr;
      wire [15:0] instruction;
      input [15:0] instruction_in;
      wire rst_n;
      wire ram_clk;
/*AUTOREG*/
/*AUTOWIRE*/
 /* simd_useq AUTO_TEMPLATE(
  ); */

simd_useq UUSEQ (
                        /*AUTOINST*/
                    // Outputs
                    .pc_r                   (pc_r[15:0]),
                                            .mem_in
                                                    (
                                               mem_in
                                               [15:0]),
                                            .mem_addr
```

```
                                                (
                                            mem_addr
                                            [15:0]),
                                        .mem_cs

                                                (
                                            mem_cs),
                                        .mem_we

                                                (
                                            mem_we),

                        // Inputs
                        .instruction        (instruction
                            [15:0]),
                                        .clk

                                                (clk
                                            ),
                        .rst_n              (rst_n),
                                        .mem_out

                                                (
                                            mem_out
                                            [15:0]));

`ifdef negativeedge

                                        mem_wrapper
                                            U_USEQ_16_RAM
                                              (
                        .read_data          (mem_out[15:0])
                            ,
                        // Inputs
                        .clk                (!clk),
                                              // Templated
```

```
                .addr                   (mem_addr
                   [15:0]), // Templated
                .write_data             (mem_in[15:0]),
                    // Templated
                .cs                     (1'b1),
                                       // Templated
                .we                     (mem_we));
`else
                                        mem_wrapper
                                           U_USEQ_16_RAM
                                             (
                .read_data              (mem_out[15:0])
                   ,
                // Inputs
                .clk                 (clk),
                                       // Templated
                .addr                   (mem_addr
                   [15:0]), // Templated
                .write_data             (mem_in[15:0]),
                    // Templated
                .cs                     (1'b1),
                                       // Templated
                .we                     (mem_we));

`endif

                                  prog_mem_wrapper
                                     PROGMEMWRAP (
                                          .load_program (
                                             load_program
                                             ),
                                          .prog_clk (
                                             prog_clk),
                                          .prog_adr (
                                             prog_adr),
                                          .clk     (clk),
                                          .pc_r    (pc_r
                                             [15:0]),
                                          .instruction_in
                                              (
                                             instruction_in
```

```
                                                    [15:0]),
                                                .instruction
                                                    (
                                                    instruction
                                                    [15:0]),
                                                .rst_n_in (
                                                    rst_n_temp),
                                                .rst_n_out (
                                                    rst_n));

endmodule // simd
=======
`timescale 1ns/1ps
//`include "simd_params.v"

module chip_top (/*AUTOARG*/
  // Outputs
  cpu_ack,
  // Inputs
  sel_mode ,rst_n_temp, cpu_req, load_program, prog_clk
     , clk,ramdsel,aludsel,rfdsel,memdsel
  );

  input rst_n_temp;                    // To U_USEQ of
     simd_useq.v
  input load_program;
  input prog_clk;
  input clk;
  input [3:0] ramdsel;
  input [3:0] aludsel;
  input [3:0] rfdsel;
        input [3:0] memdsel;

  // End of automatics
  /*AUTOOUTPUT*/
  // Beginning of automatic outputs (from unused
     autoinst outputs)
  wire [15:0] pc_r;                    // From U_USEQ of
     simd_useq.v
  output cpu_ack;
  input  cpu_req;
```

```
input [3:0] sel_mode;
// End of automatics
wire mem_cs,mem_we,mem_clk;
wire  [15:0] mem_addr;
wire [15:0] mem16_out;
wire [15:0] mem_in;
  wire [15:0] mem_out;
      wire [10:0]count;
      wire [15:0] instruction;
      wire [15:0] instruction_in;
      wire rst_n;
      wire ram_clk;
/*AUTOREG*/
/*AUTOWIRE*/
 /* simd_useq AUTO_TEMPLATE(
  ); */

simd_useq UUSEQ (
                        /*AUTOINST*/
                    // Outputs
                    .pc_r                (pc_r[15:0]),
                                         .mem_in
                                                 (
                                           mem_in
                                           [15:0]),
                                         .mem_addr
                                                 (
                                           mem_addr),
                                         .mem_cs
                                                 (
                                           mem_cs),
                                         .mem_we
                                                 (
```

```
                                    mem_we),
                                .mem_clk

                                        (
                                    mem_clk),
                                .sel_mode

                                        (
                                    sel_mode
                                    [3:0]),
                                .cpu_ack

                                        (
                                    cpu_ack),
                                .ram_clk

                                        (
                                    ram_clk),

        // Inputs
        .instruction            (instruction
            [15:0]),
                                .cpu_req

                                        (
                                    cpu_req),
        .rst_n                  (rst_n),
                                .mem_out

                                        (
                                    mem_out
                                    [15:0]),
                                .clk
```

```
                                        (clk
                                    ),
                                .ramdsel

                                        (
                                    ramdsel
                                    [3:0]),
                                .aludsel

                                        (
                                    aludsel
                                    [3:0]),
                                .rfdsel

                                        (
                                    rfdsel[3:0])
                                    ,
                                .memdsel

                                        (
                                    memdsel
                                    [3:0]));

                                mem_wrapper
                                    U_USEQ_16_RAM
                                     (
.read_data                      (mem_out[15:0])
    ,
// Inputs
.clock                          (mem_clk),
                                // Templated
.addr                           (mem_addr
   [15:0]), // Templated
.write_data                     (mem_in[15:0]),
    // Templated
```

```
                .cs                     (1'b1),
                                       // Templated
                .we                     (mem_we));


                                prog_mem_wrapper
                                  PROGMEMWRAP (
                                        .load_program (
                                          load_program
                                          ),
                                        .prog_clk (
                                          prog_clk),
                                        .prog_adr (
                                          count[10:0])
                                          ,
                                        .clk     (
                                          ram_clk),
                                        .pc_r    (pc_r
                                          [15:0]),
                                        .instruction_in
                                           (
                                          instruction_in
                                          [15:0]),
                                        .instruction
                                              (
                                          instruction
                                          [15:0]),
                                        .rst_n_in (
                                          rst_n_temp),
                                        .rst_n_out (
                                          rst_n));


endmodule // simd
```

K.J - async chip top

```
`timescale 1ns/1ps
//`include "simd_params.v"

module chip_top (/*AUTOARG*/
```

```
  // Outputs
  cpu_ack,
  // Inputs
 rst_n_temp, load_program, prog_clk, clk, ramdsel,
    aludsel, rfdsel,memdsel,sel_mode,cpu_req
  );

  input rst_n_temp;                    // To U_USEQ of
     simd_useq.v
  input load_program;
  input prog_clk;
  input clk;
  input [3:0] ramdsel;
  input [3:0] aludsel;
  input [3:0] rfdsel;
        input [3:0] memdsel;

  // End of automatics
  /*AUTOOUTPUT*/
  // Beginning of automatic outputs (from unused
     autoinst outputs)
  wire [15:0] pc_r;                    // From U_USEQ of
     simd_useq.v
  output cpu_ack;
  input  cpu_req;
  input [3:0] sel_mode;
  // End of automatics
  wire mem_cs,mem_we,mem_clk;
  wire  [15:0] mem_addr;
  wire [15:0] mem16_out;
  wire [15:0] mem_in;
    wire [15:0] mem_out;
        wire [10:0]count;
        wire [15:0] instruction;
        wire [15:0] instruction_in;
        wire rst_n;
        wire ram_clk;
  /*AUTOREG*/
  /*AUTOWIRE*/
   /* simd_useq AUTO_TEMPLATE(
    ); */
```

```
simd_useq UUSEQ (
                    /*AUTOINST*/
            // Outputs
            .pc_r                   (pc_r[15:0]),
            .mem_in
                                    (
                                       mem_in
                                       [15:0]),
            .mem_addr
                                    (
                                       mem_addr),
            .mem_cs
                                    (
                                       mem_cs),
            .mem_we
                                    (
                                       mem_we),
            .mem_clk
                                    (
                                       mem_clk),
            .ram_clk
                                    (
                                       ram_clk),
            // Inputs
            .instruction            (instruction
               [15:0]),
            .clk                    (clk),
            .rst_n                  (rst_n),
```

```
                                .mem_out

                                        (
                                    mem_out
                                    [15:0]));


                                mem_wrapper
                                    U_USEQ_16_RAM
                                     (
.read_data                      (mem_out[15:0])
    ,
// Inputs
.clock                          (mem_clk),
                                // Templated
.addr                           (mem_addr
   [15:0]), // Templated
.write_data                     (mem_in[15:0]),
    // Templated
.cs                             (1'b1),
                               // Templated
.we                             (mem_we));


                        prog_mem_wrapper
                           PROGMEMWRAP (
                                .load_program (
                                   load_program
                                   ),
                                .prog_clk (
                                   prog_clk),
                                .prog_adr (
                                   count[10:0])
                                   ,
                                .clk     (
                                   ram_clk),
                                .pc_r    (pc_r
                                   [15:0]),
```

```
                                        .instruction_in
                                            (
                                          instruction_in
                                          [15:0]),
                                        .instruction
                                              (
                                          instruction
                                          [15:0]),
                                        .rst_n_in (
                                          rst_n_temp),
                                        .rst_n_out (
                                          rst_n));


endmodule // simd
```

K.K - sync simd

```
module simd (/*AUTOARG*/
  // Outputs
  pc_r,
  // Inputs
  rst_n, instruction, clk
  );

`include "simd_params.v"
//`include "simd_features.v"


  /*AUTOINPUT*/
  // Beginning of automatic inputs (from unused
     autoinst inputs)
  input                 clk;                    // To
     U_USEQ of simd_useq.v
  input [USEQ_INST_MSB:0] instruction;          // To
     U_USEQ of simd_useq.v
  input                 rst_n;                  // To
     U_USEQ of simd_useq.v
  // End of automatics
  /*AUTOOUTPUT*/
```

```
  // Beginning of automatic outputs (from unused
     autoinst outputs)
  output [USEQ_WORD_MSB:0] pc_r;                    // From
      U_USEQ of simd_useq.v
  // End of automatics
  wire mem_cs,mem_we,mem_clk;
  wire  [USEQ_WORD_MSB:0] mem_addr;
  wire [USEQ_WORD_MSB:0] mem16_out;
  wire [USEQ_WORD_MSB:0] mem_in;
    wire [USEQ_WORD_MSB:0] mem_out;
  /*AUTOREG*/
  /*AUTOWIRE*/
   /* simd_useq AUTO_TEMPLATE(
    ); */

`ifdef NTL
        wire SI, SCAN_ENABLE,test_se, SO;
        assign SI = 0;
        assign SCAN_ENABLE=0;
        assign test_se=0;
  simd_useq U_USEQ (
                          /*AUTOINST*/
                    // Outputs
                    .pc_r                   (pc_r[
                       USEQ_WORD_MSB:0]),
                                            .mem_in

                                                    (
                                               mem_in[
                                               USEQ_WORD_MSB
                                               :0]),
                                            .mem_addr

                                                    (
                                               mem_addr),
                                            .mem_cs

                                                    (
```

```
                                    mem_cs),
                                .mem_we

                                        (
                                    mem_we),
                                .mem_clk

                                        (
                                    mem_clk),
        // Inputs
        .instruction             (instruction[
            USEQ_INST_MSB:0]),
        .clk                     (clk),
        .rst_n                   (rst_n),
                                .mem_out

                                        (
                                    mem_out[
                                    USEQ_WORD_MSB
                                    :0]),
                                .SI

                                            (SI)
                                    ,
                                .SCAN_ENABLE

                                    (SCAN_ENABLE
                                    ),
                                .test_se

                                        (
                                    test_se),
                                .SO
```

```
                                                    (SO)
                                                );
`else
simd_useq U_USEQ (
                                /*AUTOINST*/
                            // Outputs
                            .pc_r                   (pc_r[
                               USEQ_WORD_MSB:0]),
                                                    .mem_in

                                                            (
                                                       mem_in[
                                                       USEQ_WORD_MSB
                                                       :0]),
                                                    .mem_addr

                                                            (
                                                       mem_addr),
                                                    .mem_cs

                                                            (
                                                       mem_cs),
                                                    .mem_we

                                                            (
                                                       mem_we),
                                                    .mem_clk

                                                            (
                                                       mem_clk),
                            // Inputs
                            .instruction            (instruction[
                               USEQ_INST_MSB:0]),
                            .clk                    (clk),
```

```
                        .rst_n                  (rst_n),
                                                .mem_out
                                                        (
                                                    mem_out[
                                                    USEQ_WORD_MSB
                                                    :0]));
`endif
mem_wrapper U_USEQ_16_RAM (
                        .read_data              (mem_out[
                          USEQ_WORD_MSB:0]),
                        // Inputs
                        .clock                  (mem_clk),
                                                // Templated
                        .addr                   (mem_addr[
                          USEQ_WORD_MSB:0]), // Templated
                        .write_data             (mem_in[
                          USEQ_WORD_MSB:0]), // Templated
                        .cs                     (1'b1),
                                               // Templated
                        .we                     (mem_we));
/*
 ram #(.WORD_WIDTH(USEQ_WORD_BITS),
         .ADDR_WIDTH(USEQ_WORD_BITS))
  U_USEQ_16_RAM (
                        .read_data              (mem_out[
                          USEQ_WORD_MSB:0]),
                        // Inputs
                        .clock                  (mem_clk),
                                                // Templated
                        .addr                   (mem_addr[
                          USEQ_WORD_MSB:0]), // Templated
                        .write_data             (mem_in[
                          USEQ_WORD_MSB:0]), // Templated
                        .cs                     (1'b1),
                                               // Templated
                        .we                     (mem_we));

        */
   /* simd_agen AUTO_TEMPLATE(
```

```
    ); */
  simd_agen U_AGEN (
                                /*AUTOINST*/);

   /* simd_pe_array AUTO_TEMPLATE(
    ); */
  // simd_pe_array U_PE_ARRAY (
  //                             /*AUTOINST*/);




endmodule // simd
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.L - async simd

```
module simd (/*AUTOARG*/
  // Outputs
  pc_r, cpu_ack, ram_clk,
  // Inputs
  sel_mode ,rst_n, instruction, cpu_req, clk
  );

`include "simd_params.v"

  input [USEQ_INST_MSB:0] instruction;          // To
     U_USEQ of simd_useq.v
  input                   rst_n;                // To
     U_USEQ of simd_useq.v
  // End of automatics
  /*AUTOOUTPUT*/
  // Beginning of automatic outputs (from unused
     autoinst outputs)
```

```
  output [USEQ_WORD_MSB:0] pc_r;                    // From
      U_USEQ of simd_useq.v
  output ram_clk;
  output cpu_ack;
  input  cpu_req;
  input [3:0] sel_mode;
  input clk;
  // End of automatics
  wire mem_cs,mem_we,mem_clk;
  wire  [USEQ_WORD_MSB:0] mem_addr;
  wire [USEQ_WORD_MSB:0] mem16_out;
  wire [USEQ_WORD_MSB:0] mem_in;
    wire [USEQ_WORD_MSB:0] mem_out;
  /*AUTOREG*/
  /*AUTOWIRE*/
   /* simd_useq AUTO_TEMPLATE(
    ); */

`ifdef NTL
wire SI, SCAN_ENABLE,test_se, SO;
        assign SI = 0;
        assign SCAN_ENABLE=0;
        assign test_se=0;
  simd_useq U_USEQ (
                          /*AUTOINST*/
                    // Outputs
                    .pc_r                   (pc_r[
                       USEQ_WORD_MSB:0]),
                                            .mem_in

                                                    (
                                               mem_in[
                                               USEQ_WORD_MSB
                                               :0]),
                                            .mem_addr

                                                    (
                                               mem_addr),
```

```
                                    .mem_cs
                                            (
                                      mem_cs),
                                    .mem_we
                                            (
                                      mem_we),
                                    .mem_clk
                                            (
                                      mem_clk),
                                    .sel_mode
                                            (
                                      sel_mode
                                      [3:0]),
                                    .cpu_ack
                                            (
                                      cpu_ack),
                                    .ram_clk
                                            (
                                      ram_clk),
                  // Inputs
                  .instruction      (instruction[
                     USEQ_INST_MSB:0]),
                                    .cpu_req
                                            (
                                      cpu_req),
                  .rst_n            (rst_n),
```

```
                                                .mem_out
                                                        (
                                                   mem_out[
                                                   USEQ_WORD_MSB
                                                   :0]),
                                                .clk
                                                        (clk
                                                   ),
                                                .SI
                                                        (SI)
                                                   ,
                                                .SCAN_ENABLE
                                                   (SCAN_ENABLE
                                                   ),
                                                .test_se
                                                        (
                                                   test_se),
                                                .SO
                                                        (SO)
                                                   );
`else
  simd_useq U_USEQ (
                              /*AUTOINST*/
                          // Outputs
                          .pc_r                 (pc_r[
                             USEQ_WORD_MSB:0]),
                                                .mem_in
```

```
                (
    mem_in[
    USEQ_WORD_MSB
    :0]),
.mem_addr

                (
    mem_addr),
.mem_cs

                (
    mem_cs),
.mem_we

                (
    mem_we),
.mem_clk

                (
    mem_clk),
.sel_mode

                (
    sel_mode
    [3:0]),
.cpu_ack

                (
    cpu_ack),
.ram_clk

                (
    ram_clk),
```

```
                        // Inputs
                        .instruction          (instruction[
                           USEQ_INST_MSB:0]),
                                              .cpu_req


                                                    (
                                                 cpu_req),
                        .rst_n                (rst_n),
                                              .mem_out


                                                    (
                                                 mem_out[
                                                 USEQ_WORD_MSB
                                                 :0]));
`endif


                                              mem_wrapper
                                                 U_USEQ_16_RAM
                                                  (
                        .read_data            (mem_out[
                           USEQ_WORD_MSB:0]),
                        // Inputs
                        .clock                (mem_clk),
                                              // Templated
                        .addr                 (mem_addr[
                           USEQ_WORD_MSB:0]), // Templated
                        .write_data           (mem_in[
                           USEQ_WORD_MSB:0]), // Templated
                        .cs                   (1'b1),
                                             // Templated
                        .we                   (mem_we));
/*ram #(.WORD_WIDTH(USEQ_WORD_BITS),
        .ADDR_WIDTH(USEQ_WORD_BITS))
 U_USEQ_16_RAM (
                        .read_data            (mem_out[
                           USEQ_WORD_MSB:0]),
                        // Inputs
```

```
                        .clock                  (mem_clk),
                                                // Templated
                        .addr                   (mem_addr[
                          USEQ_WORD_MSB:0]), // Templated
                        .write_data             (mem_in[
                          USEQ_WORD_MSB:0]), // Templated
                        .cs                     (1'b1),
                                               // Templated
                        .we                     (mem_we));
          */
    /* simd_agen AUTO_TEMPLATE(
     ); */
   simd_agen U_AGEN (
                                /*AUTOINST*/);

    /* simd_pe_array AUTO_TEMPLATE(
     ); */
   // simd_pe_array U_PE_ARRAY (
   //                             /*AUTOINST*/);




endmodule // simd
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.M - Synchrounos CPU core

```
module simd_useq (/*AUTOARG*/
   // Outputs
   pc_r, mem_in, mem_addr, mem_cs,mem_we,
   // Inputs
   instruction, clk, rst_n,mem_out
   );
```

```
`include "simd_params.v"

  input  [USEQ_INST_MSB:0] instruction; // Instruction
     word fetched from memory

  output [USEQ_WORD_MSB:0] pc_r; // address for the
     code memory
  output mem_cs,mem_we;
  output [USEQ_WORD_MSB:0] mem_addr;
  output [USEQ_WORD_MSB:0] mem_in;


  input                    clk;
  input                    rst_n;
  input                         [USEQ_WORD_MSB:0]
            mem_out;

  /*AUTOINPUT*/
  /*AUTOOUTPUT*/

  /*AUTOREG*/
  /*AUTOWIRE*/


  // Register file
  reg [3:0]                     rf_sela;
  reg [3:0]                     rf_selb;
  reg [3:0]                     rf_selc;
  reg                           rf_write;

  wire [USEQ_WORD_MSB:0]        rf_a;
  wire [USEQ_WORD_MSB:0]        rf_b;
  wire [USEQ_WORD_MSB:0]        rf_c;

  // alu
  reg  [USEQ_WORD_MSB:0]         alu_a;
  reg  [USEQ_WORD_MSB:0]         alu_b;
  wire [USEQ_WORD_MSB:0]         alu_c;

  reg [USEQ_WORD_MSB:0]          dest_bus;
```

```
  // PC
  wire [USEQ_WORD_MSB:0]              pc_r;
  reg [USEQ_WORD_MSB:0]               npc;
  reg                                 write_to_pc;

  // Flow control
  reg                                 halt;
  reg                                 halt_r;



  // Special register support
  reg [USEQ_WORD_MSB:0]               sr_read_bus;

  // PE array address register
  // Fixme - those must be moved outside uSeq, to the
     address generator block
  // We keep them here for now for instruction &
     toolchain tests

  reg [USEQ_WORD_MSB:0]               ar0_r;
  reg [USEQ_WORD_MSB:0]               ar1_r;
  reg [USEQ_WORD_MSB:0]               ar2_r;
  reg [2:0]                           write_ar;

  assign pc_r = npc;

   /* //@begin[inst_decode]
wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE1_OFFSET +:
   USEQ_INST_FORMAT_OPCODE1_SIZE];

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd  = dec_inst_r[
   USEQ_INST_FORMAT_RD_OFFSET +:
   USEQ_INST_FORMAT_RD_SIZE];

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE3_OFFSET +:
   USEQ_INST_FORMAT_OPCODE3_SIZE];
```

```
//wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3 =
    dec_inst_r[15:12];

wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2 = dec_inst_r
    [USEQ_INST_FORMAT_RS2_OFFSET +:
    USEQ_INST_FORMAT_RS2_SIZE];

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4 = dec_inst_r
    [USEQ_INST_FORMAT_UK4_OFFSET +:
    USEQ_INST_FORMAT_UK4_SIZE];

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1 = dec_inst_r
    [USEQ_INST_FORMAT_RS1_OFFSET +:
    USEQ_INST_FORMAT_RS1_SIZE];

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4 =
    dec_inst_r[USEQ_INST_FORMAT_OPCODE4_OFFSET +:
    USEQ_INST_FORMAT_OPCODE4_SIZE];

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8 = dec_inst_r[
    USEQ_INST_FORMAT_K8_OFFSET +:
    USEQ_INST_FORMAT_K8_SIZE];

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond =
    dec_inst_r[USEQ_INST_FORMAT_COND_OFFSET +:
    USEQ_INST_FORMAT_COND_SIZE];

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12 = dec_inst_r
    [USEQ_INST_FORMAT_K12_OFFSET +:
    USEQ_INST_FORMAT_K12_SIZE];

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4 = dec_inst_r[
    USEQ_INST_FORMAT_K4_OFFSET +:
    USEQ_INST_FORMAT_K4_SIZE];

  //@end[inst_decode]
 */

wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1;

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd;
```

```
wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3;

wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2;

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4;

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1;

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4;

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8;

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond;

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12;

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4;

//@begin[sig_mux_sel]
reg [USEQ_SEL_RFB_MSB:0] dec_rf_selb_mux_sel;
reg [USEQ_SEL_ALUB_MSB:0] dec_alu_sourceb_mux_sel;
reg [USEQ_SEL_DEST_MSB:0] dec_writeback_dest_mux_sel;
reg [USEQ_SEL_PC_MSB:0] dec_pc_next_mux_sel;
reg [USEQ_SEL_RFC_MSB:0] dec_rf_selc_mux_sel;
reg [USEQ_SEL_WBSOURCE_MSB:0]
  dec_writeback_source_mux_sel;
reg [USEQ_SEL_ALUOP_MSB:0] dec_alu_op_mux_sel;
reg [USEQ_SEL_ALUA_MSB:0] dec_alu_sourcea_mux_sel;
reg [USEQ_SEL_RFA_MSB:0] dec_rf_sela_mux_sel;
reg [USEQ_FLOW_HALT_MSB:0] dec_flow_halt_mux_sel;
//@end[sig_mux_sel]

  // temporay code for address registers
  always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      /*AUTORESET*/
      // Beginning of autoreset for uninitialized flops
      ar0_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar1_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar2_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
```

```
      // End of automatics
    end
    else begin
      if(write_ar[2:0] == 3'b001)
        ar0_r <= dest_bus;
      if(write_ar[2:0] == 3'b010)
        ar1_r <= dest_bus;
      if(write_ar[2:0] == 3'b100)
        ar2_r <= dest_bus;
    end
  end


  // decoder for register file, selection for port A
  // Constants in the file rf_sela.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_sela_mux_sel)
      USEQ_SEL_RFA_R0: begin
        rf_sela <= 0;
      end
      USEQ_SEL_RFA_RS1: begin
        rf_sela <= dec_rs1;
      end
      USEQ_SEL_RFA_RD: begin
        rf_sela <= dec_rd;
      end
      default:
        rf_sela <= dec_rs1;
    endcase
  end




  // decoder for register file, selection for port B
  // Constants in the file rf_selb.generated.v (and
     useq_params.v of course)

  always @* begin
    case(dec_rf_selb_mux_sel)
```

```
            USEQ_SEL_RFB_RD: begin
                  rf_selb <= dec_rd;
            end
        USEQ_SEL_RFB_R1: begin
          rf_selb <= 'b1;
        end
        USEQ_SEL_RFB_RS1: begin
          rf_selb <= dec_rs1;
        end
        USEQ_SEL_RFB_RS2: begin
          rf_selb <= dec_rs2;
        end
        default:
          rf_selb <= dec_rs2;
    endcase
  end


  // decoder for register file, selection for port C (
     writeback)
  // Constants in the file rf_selc.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_selc_mux_sel)
      USEQ_SEL_RFC_R0: begin
        rf_selc <= 0; // Fixme : really needed ?
      end
      USEQ_SEL_RFC_R1: begin
        rf_selc <= 'b1;
      end
      USEQ_SEL_RFC_RS2: begin
        rf_selc <= dec_rs2;
      end
      USEQ_SEL_RFC_RD: begin
        rf_selc <= dec_rd;
      end
      default:
        rf_selc <= dec_rd;
    endcase
  end
```

```
// decoder for ALU port A
// see alu_sourcea.generated.v
always @* begin
  case(dec_alu_sourcea_mux_sel)
    USEQ_SEL_ALUA_RFA: begin
      alu_a <= rf_a;
    end
    USEQ_SEL_ALUA_NPC: begin
      alu_a <= npc;
    end
  default:
    alu_a <= rf_a;
  endcase
end


//event evt_dbg;

// decoder for ALU port B
// see alu_sourcea.generated.v
always @* begin
  case(dec_alu_sourceb_mux_sel)
    USEQ_SEL_ALUB_RFB: begin
      alu_b <= rf_b;
    end
    USEQ_SEL_ALUB_NOT_RFB: begin
      alu_b <= ~rf_b;
    end
    USEQ_SEL_ALUB_UK4: begin
      alu_b <= {{(USEQ_WORD_MINUS_K4){1'b0}},dec_uk4
          };
    end
    USEQ_SEL_ALUB_K4: begin // Sign extention
      alu_b <= {{(USEQ_WORD_MINUS_K4+1){dec_uk4[3]}},
         dec_uk4[2:0]};
    end
    USEQ_SEL_ALUB_K8: begin // Sign extention
      alu_b <= {{(USEQ_WORD_MINUS_K8+1){dec_k8[7]}},
         dec_k8[6:0]};
    end
```

```
        USEQ_SEL_ALUB_K12: begin // no sign extention
          alu_b <= {dec_k12,{(USEQ_WORD_MINUS_K12){1'b0
              }}};
        end

        USEQ_SEL_ALUB_RFB_OR_UK4: begin
          alu_b <= rf_b | {{(USEQ_WORD_MINUS_K4){1'b0}},
              dec_uk4};
        end
        USEQ_SEL_ALUB_NOT_RFB_OR_UK4: begin
          alu_b <= ~(rf_b | {{(USEQ_WORD_MINUS_K4){1'b0
              }},dec_uk4});
        end
        USEQ_SEL_ALUB_SR: begin
          alu_b <= sr_read_bus;
        end

        default:
          alu_b <= rf_b;
      endcase
    end


reg [USEQ_WORD_MSB:0] mem16_in ;
reg mem16_write;
reg [USEQ_WORD_MSB:0] mem16_addr ;
wire [USEQ_BYTE_MSB:0] mem8_out;
reg [USEQ_BYTE_MSB:0] mem8_in;
reg mem8_write;
reg [USEQ_WORD_MSB:0] mem8_addr;

 wire mem_cs;
  reg mem_we;
  reg  [USEQ_WORD_MSB:0] mem_addr;
  wire [USEQ_WORD_MSB:0] mem16_out;
  wire [USEQ_WORD_MSB:0] mem_in;


assign mem16_out = mem_out;
assign mem8_out = mem_out;
```

```
assign mem_in = alu_c;
assign mem_cs = 0;
  // see wb_source.generated.v
  always @* begin
    case(dec_writeback_source_mux_sel)
      USEQ_SEL_WBSOURCE_ALU: begin
        dest_bus <= alu_c;
                mem16_addr <=rf_b;
                mem8_addr <= rf_b;
                mem_addr <= rf_b;
      end
      USEQ_SEL_WBSOURCE_RFB: begin
        dest_bus <= rf_b;
                mem16_addr <=rf_b;
                mem8_addr <= rf_b;
                mem_addr <= rf_b;
                mem16_in <= alu_c;
                mem8_in <= alu_c;
      end
      USEQ_SEL_WBSOURCE_MEM16: begin
        dest_bus <= mem16_out;
                mem16_addr <=alu_c;
                mem8_addr <=alu_c;
                mem16_in <= alu_c;
                mem_addr <= alu_c;
      end
      USEQ_SEL_WBSOURCE_MEM8: begin
        dest_bus <= mem8_out;
                mem16_addr <=alu_c;
                mem8_addr <=alu_c;
                mem8_in <= alu_c;
                mem_addr <= alu_c;
      end
     default: begin
       dest_bus <= alu_c;
                mem8_addr <=rf_b;
                mem8_in <= alu_c;
                mem16_addr <=rf_b;
                mem16_in <= alu_c;
                mem_addr <= rf_b;
```

```
                end
    endcase
  end


  assign rf_c = dest_bus;
// assign mem8_in = alu_c;
// assign mem16_in = alu_c;

  // Write back - destination selection
  // see wb_dest.generated.v
  always @* begin
    rf_write = 1'b0;
    write_ar = 3'b0;
        mem8_write = 1'b0;
        mem16_write = 1'b0;
        mem_we = 1'b0;
    case(dec_writeback_dest_mux_sel)
      USEQ_SEL_DEST_RFC: begin
        rf_write = 1'b1;
      end
      USEQ_SEL_DEST_COND_LINK_REG: begin
        rf_write = (dec_cond == USEQ_COND_ALAL);  // we
            write to the link register only if the
           condition field
        // is set to USEQ_COND_ALAL (ALways And Link)
      end
      USEQ_SEL_DEST_MEM16: begin
        // Fixme
                mem16_write = 1'b1;
                mem_we = 1'b1;
      end
      USEQ_SEL_DEST_MEM8: begin
                mem8_write = 1'b1;
        mem_we = 1'b1;
                // Fixme
      end
      USEQ_SEL_DEST_SR: begin
        case(dec_rd)
         USEQ_SR_AR0 : begin
           write_ar = 3'b001;
```

```
            end
            USEQ_SR_AR1 : begin
              write_ar = 3'b010;
            end
            USEQ_SR_AR2 : begin
              write_ar = 3'b100;
            end
             default: begin
               write_ar = 3'b000;
             end

          endcase
        end
        default: begin
          rf_write = 1'b0;
          write_ar = 3'b0;
        end
      endcase
    end

    // Selection of special register for read
    // To be used by mfrs instruction

    always @* begin
      case(dec_rs1)
        USEQ_SR_AR0 : begin
          sr_read_bus = ar0_r;
        end
        USEQ_SR_AR1 : begin
          sr_read_bus = ar1_r;
        end
        USEQ_SR_AR2 : begin
          sr_read_bus = ar2_r;
        end
        USEQ_SR_SR : begin
          sr_read_bus = 0; // Fixme
        end
        default: begin
        end

      endcase
```

```
end

// Halt instruction
always @* begin
      if(rst_n == 0) begin
              halt = 1'b0;
      end else begin
  case(dec_flow_halt_mux_sel)
    USEQ_FLOW_HALT_YES: begin
      halt = 1'b1;
    end
    USEQ_FLOW_HALT_NO: begin
      halt = 1'b0;
    end
    default: begin
      halt = 1'b0;
    end
  endcase
      end
end
```

```
wire [15:0] decode_string;

assign decode_string = {dec_opcode1,dec_opcode3,
   dec_opcode4,dec_cond};
reg [39:0]          _codeop_ascii_r;          //
   Decode of state_r
initial
  _codeop_ascii_r = "Unknown";


always @* begin
  // FIXME : we need the default case
  casez(decode_string)
  //@begin[case_inst]
  USEQ_OPCODES_AND: begin
```

```
        _codeop_ascii_r = "USEQ_OPCODES_AND";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
```

```
    end
    USEQ_OPCODES_LDIR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_LDIR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R1;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K12;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_HALT: begin
        _codeop_ascii_r = "USEQ_OPCODES_HALT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
```

```
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_YES;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LB: begin
        _codeop_ascii_r = "USEQ_OPCODES_LB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM8;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
```

```
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_NOT_RFB
            ;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_RET: begin
        _codeop_ascii_r = "USEQ_OPCODES_RET";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_PC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R14;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_FROM_ALU;
    end
    USEQ_OPCODES_LW: begin
        _codeop_ascii_r = "USEQ_OPCODES_LW";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM16;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB3: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
```

```
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SUB;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BRPC: begin
        _codeop_ascii_r = "USEQ_OPCODES_BRPC";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RS2;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K8;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_NPC;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_SUB2: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
```

```
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_NOT_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ANDR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ANDR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MFSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MFSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_SR;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
```

```
USEQ_OPCODES_MTSR: begin
      _codeop_ascii_r = "USEQ_OPCODES_MTSR";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_SR;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_ORI4: begin
      _codeop_ascii_r = "USEQ_OPCODES_ORI4";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel =
         USEQ_SEL_ALUB_RFB_OR_UK4;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_SRAI: begin
      _codeop_ascii_r = "USEQ_OPCODES_SRAI";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
```

```
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SEXT: begin
        _codeop_ascii_r = "USEQ_OPCODES_SEXT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SEXT;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BR: begin
        _codeop_ascii_r = "USEQ_OPCODES_BR";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_XOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
```

```
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRA: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRA";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD3: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
```

```
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SW: begin
        _codeop_ascii_r = "USEQ_OPCODES_SW";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_MEM16;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
```

```
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SB: begin
        _codeop_ascii_r = "USEQ_OPCODES_SB";
```

```
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_MEM8
            ;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_OR: begin
        _codeop_ascii_r = "USEQ_OPCODES_OR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
   //@end[case_inst]
   default: begin

   end
   endcase
  end


  wire cond_branch;
  reg equal_flag_r;
  wire alu_equal;
  assign alu_equal = (alu_b == alu_c);

   always@(*)   begin
```

```
    casez(decode_string)
     USEQ_OPCODES_ADD2: begin
                    equal_flag_r <= alu_equal;
         end
    endcase
  end


assign cond_branch = equal_flag_r;
reg ram_delay;
reg ram_delay_done;


  always @(posedge clk or negedge rst_n) begin
   if(rst_n == 1'b0) begin
                    npc <= 0;
                    ram_delay<=0;
                    end
         else if(halt) begin
       npc <= pc_r;
    end else begin
         `ifdef negativeedge
         `else
         if(ram_delay==1) begin
                     ram_delay<=0;
         //          ram_delay_done = 1;
         //end else if (ram_delay_done==1) begin
                    //ram_delay_done <= 0;
         end else begin
         casez(decode_string)
                    USEQ_OPCODES_SW: begin
                              ram_delay<= 1;
                    end
                    USEQ_OPCODES_SB: begin
                              ram_delay<= 1;
                    end
                    USEQ_OPCODES_LB:   begin
                              ram_delay <= 1;
                    end
                    USEQ_OPCODES_LW: begin
                              ram_delay <=1;
```

```
                    end
                    default: begin
                     ram_delay<=0;
                    end
                    endcase
          end
          `endif
          if(ram_delay==0) begin

      case(dec_pc_next_mux_sel)
        USEQ_SEL_PC_INCREMENT: begin
            npc <= pc_r + USEQ_PC_INC;
        end
        USEQ_SEL_PC_FROM_ALU: begin
            npc <= dest_bus;
        end
        USEQ_SEL_PC_COND_LOAD_FROM_ALU: begin
          if(cond_branch) begin
           npc <= dest_bus;
          end
          else begin
            npc <= pc_r + USEQ_PC_INC;
          end
          end

        default: begin
          npc <= pc_r + USEQ_PC_INC;
        end

      endcase
          end else begin
                    npc <= pc_r;
                    end
                    end
end




  // Program counter management
```

```
/*    always @* begin
     if(halt | halt_r) begin
       npc = pc_r;
     end
     else if (write_to_pc) begin
       npc <= dest_bus;
     end
     else begin
       npc <= pc_r + USEQ_PC_INC;
     end */


  //end
 /* always @(posedge clk or negedge rst_n) begin
     if(rst_n == 1'b0) begin
       // Beginning of autoreset for uninitialized flops
       pc_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
           ram_delay<=0;
           ram_delay_done <= 0;
       // End of automatics
     end
   else begin
         if(ram_delay==1) begin
                 ram_delay<=0;
         //      ram_delay_done = 1;
         //end else if (ram_delay_done==1) begin
                 //ram_delay_done <= 0;
         end else begin

         casez(decode_string)
                 USEQ_OPCODES_SW: begin
                         ram_delay <= 1;
                 end
                 USEQ_OPCODES_SB: begin
                         ram_delay <= 1;
                 end
                 USEQ_OPCODES_LB:  begin
                         ram_delay <= 1;
                 end
                 USEQ_OPCODES_LW: begin
                         ram_delay <=1;
```

```
                end
                default: begin
                 ram_delay<=0;
                end
                endcase
        end

        if(ram_delay==0) begin
          pc_r <= npc;
        end
          halt_r <= halt;
        end
end // always @ (posedge clk or negedge rst_n)
*/


assign carry_in = 0; // FIXME

simd_useq_alu U_ALU (
                     // Outputs
                     .c                (alu_c[
                        USEQ_WORD_MSB:0]),
                     // Inputs
                     .a                (alu_a[
                        USEQ_WORD_MSB:0]),
                     .b                (alu_b[
                        USEQ_WORD_MSB:0]),
                     .op               (
                        dec_alu_op_mux_sel[
                        USEQ_SEL_ALUOP_MSB:0]),
                     .carry_in         (carry_in),
                     .carry_out        (carry_out));



 /* simd_useq_rf AUTO_TEMPLATE(
  ); */
simd_useq_rf U_RF (
                   .rf_a               (rf_a[
                      USEQ_WORD_MSB:0]),
```

```
                        .rf_b                   (rf_b[
                          USEQ_WORD_MSB:0]),
                        /*AUTOINST*/
                        // Inputs
                        .rf_sela                (rf_sela[
                          USEQ_RF_SEL_MSB:0]),
                        .rf_selb                (rf_selb[
                          USEQ_RF_SEL_MSB:0]),
                        .rf_selc                (rf_selc[
                          USEQ_RF_SEL_MSB:0]),
                        .rf_c                   (rf_c[
                          USEQ_WORD_MSB:0]),
                        .rf_write               (rf_write),
                        .clk                    (clk),
                        .rst_n                  (rst_n));



/* always @(posedge clk or negedge rst_n) begin
   if(rst_n == 1'b0) begin
     //AUTORESET
     // Beginning of autoreset for uninitialized flops
     dec_inst_r <= {(1+(USEQ_INST_MSB)){1'b0}};
     // End of automatics
   end
   else begin
     dec_inst_r <= instruction;

   end
 end
 */



simd_useq_decoder U_DC3 (.dec_opcode1          (
   dec_opcode1[USEQ_INST_FORMAT_OPCODE1_MSB:0]),

                                                .
                                                   dec_rd
```

```
(
dec_rd
[
USEQ_INST_FO
:0])
,

.
dec_opcode3

(
dec_opcode3
[
USEQ_INST_FO
:0])
,

.
dec_rs2

(
dec_rs2
[
USEQ_INST_FO
:0])
,

.
dec_uk4

(
dec_uk4
[
USEQ_INST_FO
```

```
:0])
,

.
dec_rs1

(
dec_rs1
[
USEQ_INST_FO
:0])
,

.
dec_opcode4

(
dec_opcode4
[
USEQ_INST_FO
:0])
,

.
dec_k8

(
dec_k8
[
USEQ_INST_FO
:0])
,
```

.
dec_cond

(
dec_cond
[
USEQ_INST_FO
:0])
,

.
dec_k12

(
dec_k12
[
USEQ_INST_FC
:0])
,

.
dec_k4

(
dec_k4
[
USEQ_INST_FO
:0])
,

.
instruction

(

```
instruction
[
USEQ_INST_MS
:0])
,

.
clk

(
clk
)
,

.
rst_n

(
rst_n
)
)
;

endmodule // simd_useq
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.N - Asynchronous ring req CPU core

```
module pulse_generator(out,in);
        input in;
        output out;
        wire a;
        delay_cell D1(a,in);
        xor(out, a, in);
endmodule

module delay_cell (z,i);
        parameter DELAY = 1;
        input i;
        output z;
        assign #DELAY z = i;
endmodule


module simd_useq (/*AUTOARG*/
  // Outputs
  pc_r, cpu_ack, ram_clk,
  // Inputs
  instruction, cpu_req, clk, rst_n
  );

`include "simd_params.v"

  input  [USEQ_INST_MSB:0] instruction; // Instruction
     word fetched from memory
  output [USEQ_WORD_MSB:0] pc_r; // address for the
     code memory


  input                      clk;
  input                      rst_n;
  input                    cpu_req;
  output                 cpu_ack;
  output                                    ram_clk
     ;

  /*AUTOINPUT*/
```

```
/*AUTOOUTPUT*/

/*AUTOREG*/
/*AUTOWIRE*/


reg [USEQ_WORD_MSB:0]     dec_inst_r;


// Register file
reg [3:0]                 rf_sela;
reg [3:0]                 rf_selb;
reg [3:0]                 rf_selc;
reg                       rf_write;

wire [USEQ_WORD_MSB:0]    rf_a;
wire [USEQ_WORD_MSB:0]    rf_b;
wire [USEQ_WORD_MSB:0]    rf_c;

// alu
reg  [USEQ_WORD_MSB:0]    alu_a;
reg  [USEQ_WORD_MSB:0]    alu_b;
wire [USEQ_WORD_MSB:0]    alu_c;

reg [USEQ_WORD_MSB:0]     dest_bus;


// PC
reg [USEQ_WORD_MSB:0]     pc_r;
reg [USEQ_WORD_MSB:0]     npc;
reg                       write_to_pc;

// Flow control
reg                       halt;
reg                       halt_r;




// Special register support
reg [USEQ_WORD_MSB:0]     sr_read_bus;
```

```
  reg ram_req;
  reg ram_ack;
  reg alu_req;
  reg rf_req;
  reg cpu_ack;
wire ram_clk;

  // PE array address register
  // Fixme - those must be moved outside uSeq, to the
     address generator block
  // We keep them here for now for instruction &
     toolchain tests

  reg [USEQ_WORD_MSB:0]              ar0_r;
  reg [USEQ_WORD_MSB:0]              ar1_r;
  reg [USEQ_WORD_MSB:0]              ar2_r;
  reg [2:0]                          write_ar;


//@begin[sig_mux_sel]
reg [USEQ_SEL_RFB_MSB:0]  dec_rf_selb_mux_sel;
reg [USEQ_SEL_ALUB_MSB:0]  dec_alu_sourceb_mux_sel;
reg [USEQ_SEL_DEST_MSB:0]  dec_writeback_dest_mux_sel;
reg [USEQ_SEL_PC_MSB:0]  dec_pc_next_mux_sel;
reg [USEQ_SEL_RFC_MSB:0]  dec_rf_selc_mux_sel;
reg [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel;
reg [USEQ_SEL_ALUOP_MSB:0]  dec_alu_op_mux_sel;
reg [USEQ_SEL_ALUA_MSB:0]  dec_alu_sourcea_mux_sel;
reg [USEQ_SEL_RFA_MSB:0]  dec_rf_sela_mux_sel;
reg [USEQ_FLOW_HALT_MSB:0]  dec_flow_halt_mux_sel;
//@end[sig_mux_sel]




  //@begin[inst_decode]
wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE1_OFFSET +:
```

```
    USEQ_INST_FORMAT_OPCODE1_SIZE];

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd = dec_inst_r[
    USEQ_INST_FORMAT_RD_OFFSET +:
    USEQ_INST_FORMAT_RD_SIZE];

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3 =
    dec_inst_r[USEQ_INST_FORMAT_OPCODE3_OFFSET +:
    USEQ_INST_FORMAT_OPCODE3_SIZE];

wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2 = dec_inst_r
    [USEQ_INST_FORMAT_RS2_OFFSET +:
    USEQ_INST_FORMAT_RS2_SIZE];

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4 = dec_inst_r
    [USEQ_INST_FORMAT_UK4_OFFSET +:
    USEQ_INST_FORMAT_UK4_SIZE];

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1 = dec_inst_r
    [USEQ_INST_FORMAT_RS1_OFFSET +:
    USEQ_INST_FORMAT_RS1_SIZE];

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4 =
    dec_inst_r[USEQ_INST_FORMAT_OPCODE4_OFFSET +:
    USEQ_INST_FORMAT_OPCODE4_SIZE];

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8 = dec_inst_r[
    USEQ_INST_FORMAT_K8_OFFSET +:
    USEQ_INST_FORMAT_K8_SIZE];

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond =
    dec_inst_r[USEQ_INST_FORMAT_COND_OFFSET +:
    USEQ_INST_FORMAT_COND_SIZE];

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12 = dec_inst_r
    [USEQ_INST_FORMAT_K12_OFFSET +:
    USEQ_INST_FORMAT_K12_SIZE];

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4 = dec_inst_r[
    USEQ_INST_FORMAT_K4_OFFSET +:
    USEQ_INST_FORMAT_K4_SIZE];
```

```
//@end[inst_decode]



// temporay code for address registers
always @(rf_ack or negedge rst_n) begin
  if(rst_n == 1'b0) begin
    /*AUTORESET*/
    // Beginning of autoreset for uninitialized flops
    ar0_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
    ar1_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
    ar2_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
    // End of automatics
  end
  else begin
    if(write_ar[2:0] == 3'b001)
      ar0_r <= dest_bus;
    if(write_ar[2:0] == 3'b010)
      ar1_r <= dest_bus;
    if(write_ar[2:0] == 3'b100)
      ar2_r <= dest_bus;
  end
end




always @(ram_ack or negedge rst_n) begin
  if(rst_n == 1'b0) begin
    /*AUTORESET*/
    // Beginning of autoreset for uninitialized flops
    dec_inst_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
    // End of automatics
      alu_req=0;
  end
  else begin
    dec_inst_r <= instruction;
      alu_req <= !alu_req;
```

```
    end
  end
```

```
  // decoder for register file, selection for port A
  // Constants in the file rf_sela.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_sela_mux_sel)
      USEQ_SEL_RFA_R0: begin
        rf_sela <= 0;
      end
      USEQ_SEL_RFA_RS1: begin
        rf_sela <= dec_rs1;
      end
      USEQ_SEL_RFA_RD: begin
        rf_sela <= dec_rd;
      end
      default:
        rf_sela <= dec_rs1;
    endcase
  end
```

```
  // decoder for register file, selection for port B
  // Constants in the file rf_selb.generated.v (and
     useq_params.v of course)

  always @* begin
    case(dec_rf_selb_mux_sel)
      USEQ_SEL_RFB_R1: begin
        rf_selb <= 'b1;
      end
      USEQ_SEL_RFB_RS1: begin
        rf_selb <= dec_rs1;
      end
```

```
      USEQ_SEL_RFB_RS2: begin
        rf_selb <= dec_rs2;
      end
      default:
        rf_selb <= dec_rs2;
    endcase
  end


  // decoder for register file, selection for port C (
     writeback)
  // Constants in the file rf_selc.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_selc_mux_sel)
      USEQ_SEL_RFC_R0: begin
        rf_selc <= 0; // Fixme : really needed ?
      end
      USEQ_SEL_RFC_R1: begin
        rf_selc <= 'b1;
      end
      USEQ_SEL_RFC_RS2: begin
        rf_selc <= dec_rs2;
      end
      USEQ_SEL_RFC_RD: begin
        rf_selc <= dec_rd;
      end
      default:
        rf_selc <= dec_rd;
    endcase
  end


  // decoder for ALU port A
  // see alu_sourcea.generated.v
  always @* begin
    case(dec_alu_sourcea_mux_sel)
      USEQ_SEL_ALUA_RFA: begin
        alu_a <= rf_a;
      end
      USEQ_SEL_ALUA_NPC: begin
```

```
        alu_a <= npc;
      end
    default:
      alu_a <= rf_a;
    endcase
  end


  event evt_dbg;

  // decoder for ALU port B
  // see alu_sourcea.generated.v
  always @* begin
    case(dec_alu_sourceb_mux_sel)
      USEQ_SEL_ALUB_RFB: begin
        alu_b <= rf_b;
      end
      USEQ_SEL_ALUB_NOT_RFB: begin
        alu_b <= ~rf_b;
      end
      USEQ_SEL_ALUB_UK4: begin
        alu_b <= {{(USEQ_WORD_MINUS_K4){1'b0}},dec_uk4
            };
      end
      USEQ_SEL_ALUB_K4: begin // Sign extention
        alu_b <= {{(USEQ_WORD_MINUS_K4+1){dec_uk4[3]}},
            dec_uk4[2:0]};
      end
      USEQ_SEL_ALUB_K8: begin // Sign extention
        alu_b <= {{(USEQ_WORD_MINUS_K8+1){dec_k8[7]}},
            dec_k8[6:0]};
      end
      USEQ_SEL_ALUB_K12: begin // no sign extention
        alu_b <= {dec_k12,{(USEQ_WORD_MINUS_K12){1'b0
            }}};
      end

      USEQ_SEL_ALUB_RFB_OR_UK4: begin
        alu_b <= rf_b | {{(USEQ_WORD_MINUS_K4){1'b0}},
            dec_uk4};
      end
```

```
      USEQ_SEL_ALUB_NOT_RFB_OR_UK4: begin
        alu_b <= ~(rf_b | {{(USEQ_WORD_MINUS_K4){1'b0
            }},dec_uk4});
      end
      USEQ_SEL_ALUB_SR: begin
        alu_b <= sr_read_bus;
      end

      default:
        alu_b <= rf_b;
    endcase
  end

  // Write back - source selection
  // see wb_source.generated.v
  always @* begin
    case(dec_writeback_source_mux_sel)
      USEQ_SEL_WBSOURCE_ALU: begin
        dest_bus <= alu_c;
      end
      USEQ_SEL_WBSOURCE_RFB: begin
        dest_bus <= rf_b;
      end
      USEQ_SEL_WBSOURCE_MEM16: begin
        // Fixme
      end
      USEQ_SEL_WBSOURCE_MEM8: begin
        // Fixme
      end
     default:
       dest_bus <= alu_c;
    endcase
  end



  assign rf_c = dest_bus;

  // Write back - destination selection
  // see wb_dest.generated.v
  always @* begin
```

```
    rf_write = 1'b0;
    write_ar = 3'b0;
    case(dec_writeback_dest_mux_sel)
      USEQ_SEL_DEST_RFC: begin
        rf_write = 1'b1;
      end
      USEQ_SEL_DEST_COND_LINK_REG: begin
        rf_write = (dec_cond == USEQ_COND_ALAL);  // we
            write to the link register only if the
           condition field
        // is set to USEQ_COND_ALAL (ALways And Link)
      end
      USEQ_SEL_DEST_MEM16: begin
        // Fixme
      end
      USEQ_SEL_DEST_MEM8: begin
        // Fixme
      end
      USEQ_SEL_DEST_SR: begin
        case(dec_rd)
         USEQ_SR_AR0 : begin
           write_ar = 3'b001;
         end
         USEQ_SR_AR1 : begin
           write_ar = 3'b010;
         end
         USEQ_SR_AR2 : begin
           write_ar = 3'b100;
         end
          default: begin
            write_ar = 3'b000;
          end

        endcase
      end
      default: begin
        rf_write = 1'b0;
        write_ar = 3'b0;
      end
    endcase
  end
```

```
// Selection of special register for read
// To be used by mfrs instruction

always @* begin
  case(dec_rs1)
    USEQ_SR_AR0 : begin
      sr_read_bus = ar0_r;
    end
    USEQ_SR_AR1 : begin
      sr_read_bus = ar1_r;
    end
    USEQ_SR_AR2 : begin
      sr_read_bus = ar2_r;
    end
    USEQ_SR_SR : begin
      sr_read_bus = 0; // Fixme
    end
    default: begin
    end

  endcase
end

// Halt instruction
always @* begin
  case(dec_flow_halt_mux_sel)
    USEQ_FLOW_HALT_YES: begin
      halt = 1'b1;
    end
    USEQ_FLOW_HALT_NO: begin
      halt = 1'b0;
    end
    default: begin
      halt = 1'b0;
    end
  endcase
end
```

```
wire [15:0] decode_string;

assign decode_string = {dec_opcode1,dec_opcode3,
   dec_opcode4,dec_cond};
reg [39:0]           _codeop_ascii_r;          //
   Decode of state_r
initial
  _codeop_ascii_r = "Unknown";


always @* begin
  // FIXME : we need the default case
  casez(decode_string)
  //@begin[case_inst]
  USEQ_OPCODES_AND: begin
       _codeop_ascii_r = "USEQ_OPCODES_AND";
       dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
       dec_writeback_source_mux_sel =
          USEQ_SEL_WBSOURCE_ALU;
       dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
       dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
       dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
       dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
       dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
       dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
       dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
       dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
  end
  USEQ_OPCODES_XNORR1: begin
       _codeop_ascii_r = "USEQ_OPCODES_XNORR1";
       dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
       dec_writeback_source_mux_sel =
          USEQ_SEL_WBSOURCE_ALU;
       dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
       dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
       dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
```

```
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LDIR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_LDIR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R1;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K12;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
```

```
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_HALT: begin
        _codeop_ascii_r = "USEQ_OPCODES_HALT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_YES;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LB: begin
        _codeop_ascii_r = "USEQ_OPCODES_LB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
```

```
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM8;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_NOT_RFB
            ;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_RET: begin
        _codeop_ascii_r = "USEQ_OPCODES_RET";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_PC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R14;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_FROM_ALU;
    end
    USEQ_OPCODES_LW: begin
```

```
        _codeop_ascii_r = "USEQ_OPCODES_LW";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_MEM16;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB3: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SUB;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
```

```
USEQ_OPCODES_BRPC: begin
      _codeop_ascii_r = "USEQ_OPCODES_BRPC";
      dec_writeback_dest_mux_sel =
         USEQ_SEL_DEST_COND_LINK_REG;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RS2;
      dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K8;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_NPC;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel =
         USEQ_SEL_PC_COND_LOAD_FROM_ALU;
end
USEQ_OPCODES_SUB2: begin
      _codeop_ascii_r = "USEQ_OPCODES_SUB2";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel =
         USEQ_SEL_ALUB_NOT_RFB_OR_UK4;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_ANDR1: begin
      _codeop_ascii_r = "USEQ_OPCODES_ANDR1";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel =
         USEQ_SEL_ALUB_RFB_OR_UK4;
```

```
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MFSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MFSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_SR;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MTSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MTSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_SR;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORI4: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORI4";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
```

```
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRAI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRAI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SEXT: begin
        _codeop_ascii_r = "USEQ_OPCODES_SEXT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SEXT;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BR: begin
        _codeop_ascii_r = "USEQ_OPCODES_BR";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
```

```
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
           USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_XOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRA: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRA";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD3: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD3";
```

```
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
```

```
USEQ_OPCODES_SW: begin
    _codeop_ascii_r = "USEQ_OPCODES_SW";
    dec_writeback_dest_mux_sel =
      USEQ_SEL_DEST_MEM16;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_RFB;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
    dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
    dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
    dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_SRL: begin
    _codeop_ascii_r = "USEQ_OPCODES_SRL";
    dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_ALU;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
    dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
    dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
    dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_SLL: begin
    _codeop_ascii_r = "USEQ_OPCODES_SLL";
    dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_ALU;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
    dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
    dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
```

```
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SB: begin
        _codeop_ascii_r = "USEQ_OPCODES_SB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_MEM8
            ;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_OR: begin
        _codeop_ascii_r = "USEQ_OPCODES_OR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
```

```
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
   //@end[case_inst]
   endcase
  end


  wire cond_branch;
  assign cond_branch = 1'b1; // Fixme


  always @* begin
  if(halt | halt_r) begin
      npc = pc_r;
    end
        else begin

    case(dec_pc_next_mux_sel)
      USEQ_SEL_PC_INCREMENT: begin
          npc <= pc_r + USEQ_PC_INC;
      end
      USEQ_SEL_PC_FROM_ALU: begin
          npc <= dest_bus;
      end
      USEQ_SEL_PC_COND_LOAD_FROM_ALU: begin
        if(cond_branch) begin
         npc <= dest_bus;
        end
        else begin
          npc <= pc_r + USEQ_PC_INC;
        end
      end

      default: begin
        npc <= pc_r + USEQ_PC_INC;
      end

    endcase
end
```

```
  end




  // Program counter management
/*    always @* begin
     if(halt | halt_r) begin
       npc = pc_r;
     end
     else if (write_to_pc) begin
       npc <= dest_bus;
     end
     else begin
       npc <= pc_r + USEQ_PC_INC;
     end */


  //end

  always @(ram_req or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      /*AUTORESET*/
      // Beginning of autoreset for uninitialized flops
      pc_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      // End of automatics
   rf_req<=0 ;
   cpu_ack <=0 ;
    end
    else begin
      pc_r <= npc;
      halt_r <= halt;
    end
  end // always @ (posedge clk or negedge rst_n)


  assign carry_in = 0; // FIXME


  //RAM Read
```

```
  pulse_generator PG1(ram_clk,ram_req);
  always @(cpu_req or rf_ack or negedge rst_n) begin
        if(rst_n == 1'b0) begin
                ram_req<=0;
        end
        else begin
                ram_req <= !ram_req;
        end
  end

 wire ram_req_delayed;

 delay_cell #(.DELAY (10)) D1 (ram_req_delayed,ram_req)
    ;
  always @(ram_req_delayed or negedge rst_n) begin
        if(rst_n == 1'b0) begin
                ram_ack <= 0;
        end
        else begin
                ram_ack <= !ram_ack;
        end
  end

always @(rf_ack or negedge rst_n) begin
        if(rst_n == 1'b0) begin
                ram_req <= 0;
        end
        else begin
        ram_req <= !ram_req;
        end
end

 wire rf_req_delayed;

delay_cell #(.DELAY (10)) D2 (rf_req_delayed,rf_req);
  always @(rf_req_delayed or negedge rst_n) begin
        if(rst_n == 1'b0) begin
                rf_ack <= 0;
        end
```

```
        else begin
                rf_ack <= !rf_ack;
        end
end

wire rf_clk;
pulse_generator PG1(rf_clk,rf_req);
always @(cpu_req or alu_ack or negedge rst_n) begin
        if(rst_n == 1'b0) begin
                rf_req <=0;
        end
        else begin
                rf_req <= !rf_req;
        end
end

simd_useq_alu U_ALU (
                        // Outputs
                        .c                  (alu_c[
                           USEQ_WORD_MSB:0]),
                         .alu_ack           (alu_ack),
                        // Inputs
                         .alu_req           (alu_req),
                        .a                  (alu_a[
                           USEQ_WORD_MSB:0]),
                        .b                  (alu_b[
                           USEQ_WORD_MSB:0]),
                        .op                 (
                           dec_alu_op_mux_sel[
                           USEQ_SEL_ALUOP_MSB:0]),
                        .carry_in           (carry_in),
                        .carry_out          (carry_out),
                         .rst_n             (rst_n));



 /* simd_useq_rf AUTO_TEMPLATE(
  ); */
simd_useq_rf U_RF (
                      .rf_a                 (rf_a[
                         USEQ_WORD_MSB:0]),
```

```
                .rf_b                   (rf_b[
                  USEQ_WORD_MSB:0]),
                /*AUTOINST*/
                // Inputs
                .rf_sela                (rf_sela[
                  USEQ_RF_SEL_MSB:0]),
                .rf_selb                (rf_selb[
                  USEQ_RF_SEL_MSB:0]),
                .rf_selc                (rf_selc[
                  USEQ_RF_SEL_MSB:0]),
                .rf_c                   (rf_c[
                  USEQ_WORD_MSB:0]),
                .rf_write               (rf_write),
                .clk                    (rf_clk),
                .rst_n                  (rst_n));



endmodule // simd_useq
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.O - Asynchronous simple muller pipeline CPU core

```
module simd_useq (/*AUTOARG*/
  // Outputs
  pc_r, mem_in, mem_addr, mem_cs,mem_we,
  // Inputs
  instruction, clk, rst_n,mem_out
  );

`include "simd_params.v"

  input  [USEQ_INST_MSB:0] instruction; // Instruction
     word fetched from memory
```

```
output [USEQ_WORD_MSB:0] pc_r; // address for the
   code memory
output mem_cs,mem_we;
output [USEQ_WORD_MSB:0] mem_addr;
output [USEQ_WORD_MSB:0] mem_in;


input                    clk;
input                    rst_n;
input                         [USEQ_WORD_MSB:0]
          mem_out;

/*AUTOINPUT*/
/*AUTOOUTPUT*/

/*AUTOREG*/
/*AUTOWIRE*/


// Register file
reg [3:0]                rf_sela;
reg [3:0]                rf_selb;
reg [3:0]                rf_selc;
reg                      rf_write;

wire [USEQ_WORD_MSB:0]   rf_a;
wire [USEQ_WORD_MSB:0]   rf_b;
wire [USEQ_WORD_MSB:0]   rf_c;

// alu
reg  [USEQ_WORD_MSB:0]    alu_a;
reg  [USEQ_WORD_MSB:0]    alu_b;
wire [USEQ_WORD_MSB:0]    alu_c;

reg [USEQ_WORD_MSB:0]     dest_bus;


// PC
wire [USEQ_WORD_MSB:0]      pc_r;
reg [USEQ_WORD_MSB:0]      npc;
```

```
  reg                                   write_to_pc;

  // Flow control
  reg                                   halt;
  reg                                   halt_r;




  // Special register support
  reg [USEQ_WORD_MSB:0]                 sr_read_bus;

  // PE array address register
  // Fixme - those must be moved outside uSeq, to the
     address generator block
  // We keep them here for now for instruction &
     toolchain tests

  reg [USEQ_WORD_MSB:0]                 ar0_r;
  reg [USEQ_WORD_MSB:0]                 ar1_r;
  reg [USEQ_WORD_MSB:0]                 ar2_r;
  reg [2:0]                             write_ar;

  assign pc_r = npc;

   /* //@begin[inst_decode]
wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE1_OFFSET +:
   USEQ_INST_FORMAT_OPCODE1_SIZE];

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd  = dec_inst_r[
   USEQ_INST_FORMAT_RD_OFFSET +:
   USEQ_INST_FORMAT_RD_SIZE];

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE3_OFFSET +:
   USEQ_INST_FORMAT_OPCODE3_SIZE];
//wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3  =
   dec_inst_r[15:12];
```

```
wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2  = dec_inst_r
   [USEQ_INST_FORMAT_RS2_OFFSET +:
   USEQ_INST_FORMAT_RS2_SIZE];

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4  = dec_inst_r
   [USEQ_INST_FORMAT_UK4_OFFSET +:
   USEQ_INST_FORMAT_UK4_SIZE];

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1  = dec_inst_r
   [USEQ_INST_FORMAT_RS1_OFFSET +:
   USEQ_INST_FORMAT_RS1_SIZE];

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4  =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE4_OFFSET +:
   USEQ_INST_FORMAT_OPCODE4_SIZE];

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8  = dec_inst_r[
   USEQ_INST_FORMAT_K8_OFFSET +:
   USEQ_INST_FORMAT_K8_SIZE];

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond  =
   dec_inst_r[USEQ_INST_FORMAT_COND_OFFSET +:
   USEQ_INST_FORMAT_COND_SIZE];

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12  = dec_inst_r
   [USEQ_INST_FORMAT_K12_OFFSET +:
   USEQ_INST_FORMAT_K12_SIZE];

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4  = dec_inst_r[
   USEQ_INST_FORMAT_K4_OFFSET +:
   USEQ_INST_FORMAT_K4_SIZE];

  //@end[inst_decode]
 */

wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1;

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd;

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3;
```

```
wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2;

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4;

wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1;

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4;

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8;

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond;

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12;

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4;

//@begin[sig_mux_sel]
reg [USEQ_SEL_RFB_MSB:0]  dec_rf_selb_mux_sel;
reg [USEQ_SEL_ALUB_MSB:0]  dec_alu_sourceb_mux_sel;
reg [USEQ_SEL_DEST_MSB:0]  dec_writeback_dest_mux_sel;
reg [USEQ_SEL_PC_MSB:0]  dec_pc_next_mux_sel;
reg [USEQ_SEL_RFC_MSB:0]  dec_rf_selc_mux_sel;
reg [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel;
reg [USEQ_SEL_ALUOP_MSB:0]  dec_alu_op_mux_sel;
reg [USEQ_SEL_ALUA_MSB:0]  dec_alu_sourcea_mux_sel;
reg [USEQ_SEL_RFA_MSB:0]  dec_rf_sela_mux_sel;
reg [USEQ_FLOW_HALT_MSB:0]  dec_flow_halt_mux_sel;
//@end[sig_mux_sel]

  // temporay code for address registers
  always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      /*AUTORESET*/
      // Beginning of autoreset for uninitialized flops
      ar0_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar1_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar2_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      // End of automatics
    end
    else begin
```

```
      if(write_ar[2:0] == 3'b001)
        ar0_r <= dest_bus;
      if(write_ar[2:0] == 3'b010)
        ar1_r <= dest_bus;
      if(write_ar[2:0] == 3'b100)
        ar2_r <= dest_bus;
    end
  end


  // decoder for register file, selection for port A
  // Constants in the file rf_sela.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_sela_mux_sel)
      USEQ_SEL_RFA_R0: begin
        rf_sela <= 0;
      end
      USEQ_SEL_RFA_RS1: begin
        rf_sela <= dec_rs1;
      end
      USEQ_SEL_RFA_RD: begin
        rf_sela <= dec_rd;
      end
      default:
        rf_sela <= dec_rs1;
    endcase
  end




  // decoder for register file, selection for port B
  // Constants in the file rf_selb.generated.v (and
     useq_params.v of course)

  always @* begin
    case(dec_rf_selb_mux_sel)
          USEQ_SEL_RFB_RD: begin
                rf_selb <= dec_rd;
          end
```

```
      USEQ_SEL_RFB_R1: begin
        rf_selb <= 'b1;
      end
      USEQ_SEL_RFB_RS1: begin
        rf_selb <= dec_rs1;
      end
      USEQ_SEL_RFB_RS2: begin
        rf_selb <= dec_rs2;
      end
      default:
        rf_selb <= dec_rs2;
    endcase
  end


  // decoder for register file, selection for port C (
     writeback)
  // Constants in the file rf_selc.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_selc_mux_sel)
      USEQ_SEL_RFC_R0: begin
        rf_selc <= 0; // Fixme : really needed ?
      end
      USEQ_SEL_RFC_R1: begin
        rf_selc <= 'b1;
      end
      USEQ_SEL_RFC_RS2: begin
        rf_selc <= dec_rs2;
      end
      USEQ_SEL_RFC_RD: begin
        rf_selc <= dec_rd;
      end
      default:
        rf_selc <= dec_rd;
    endcase
  end


  // decoder for ALU port A
  // see alu_sourcea.generated.v
```

```
always @* begin
  case(dec_alu_sourcea_mux_sel)
    USEQ_SEL_ALUA_RFA: begin
      alu_a <= rf_a;
    end
    USEQ_SEL_ALUA_NPC: begin
      alu_a <= npc;
    end
  default:
    alu_a <= rf_a;
  endcase
end


//event evt_dbg;

// decoder for ALU port B
// see alu_sourcea.generated.v
always @* begin
  case(dec_alu_sourceb_mux_sel)
    USEQ_SEL_ALUB_RFB: begin
      alu_b <= rf_b;
    end
    USEQ_SEL_ALUB_NOT_RFB: begin
      alu_b <= ~rf_b;
    end
    USEQ_SEL_ALUB_UK4: begin
      alu_b <= {{(USEQ_WORD_MINUS_K4){1'b0}},dec_uk4
          };
    end
    USEQ_SEL_ALUB_K4: begin // Sign extention
      alu_b <= {{(USEQ_WORD_MINUS_K4+1){dec_uk4[3]}},
          dec_uk4[2:0]};
    end
    USEQ_SEL_ALUB_K8: begin // Sign extention
      alu_b <= {{(USEQ_WORD_MINUS_K8+1){dec_k8[7]}},
          dec_k8[6:0]};
    end
    USEQ_SEL_ALUB_K12: begin // no sign extention
      alu_b <= {dec_k12,{(USEQ_WORD_MINUS_K12){1'b0
          }}};
```

```
        end

        USEQ_SEL_ALUB_RFB_OR_UK4: begin
          alu_b <= rf_b | {{(USEQ_WORD_MINUS_K4){1'b0}},
             dec_uk4};
        end
        USEQ_SEL_ALUB_NOT_RFB_OR_UK4: begin
          alu_b <= ~(rf_b | {{(USEQ_WORD_MINUS_K4){1'b0
             }},dec_uk4});
        end
        USEQ_SEL_ALUB_SR: begin
          alu_b <= sr_read_bus;
        end

        default:
          alu_b <= rf_b;
      endcase
    end


reg [USEQ_WORD_MSB:0] mem16_in ;
reg mem16_write;
reg [USEQ_WORD_MSB:0] mem16_addr ;
wire [USEQ_BYTE_MSB:0] mem8_out;
reg [USEQ_BYTE_MSB:0] mem8_in;
reg mem8_write;
reg [USEQ_WORD_MSB:0] mem8_addr;

 wire mem_cs;
  reg mem_we;
  reg  [USEQ_WORD_MSB:0] mem_addr;
  wire [USEQ_WORD_MSB:0] mem16_out;
  wire [USEQ_WORD_MSB:0] mem_in;


assign mem16_out = mem_out;
assign mem8_out = mem_out;


assign mem_in = alu_c;
assign mem_cs = 0;
```

```
// see wb_source.generated.v
always @* begin
  case(dec_writeback_source_mux_sel)
    USEQ_SEL_WBSOURCE_ALU: begin
      dest_bus <= alu_c;
               mem16_addr <=rf_b;
               mem8_addr <= rf_b;
               mem_addr <= rf_b;
    end
    USEQ_SEL_WBSOURCE_RFB: begin
      dest_bus <= rf_b;
               mem16_addr <=rf_b;
               mem8_addr <= rf_b;
               mem_addr <= rf_b;
               mem16_in <= alu_c;
               mem8_in <= alu_c;
    end
    USEQ_SEL_WBSOURCE_MEM16: begin
      dest_bus <= mem16_out;
               mem16_addr <=alu_c;
               mem8_addr <=alu_c;
               mem16_in <= alu_c;
               mem_addr <= alu_c;
    end
    USEQ_SEL_WBSOURCE_MEM8: begin
      dest_bus <= mem8_out;
               mem16_addr <=alu_c;
               mem8_addr <=alu_c;
               mem8_in <= alu_c;
               mem_addr <= alu_c;
    end
   default: begin
     dest_bus <= alu_c;
               mem8_addr <=rf_b;
               mem8_in <= alu_c;
               mem16_addr <=rf_b;
               mem16_in <= alu_c;
               mem_addr <= rf_b;
               end
  endcase
end
```

```
  assign rf_c = dest_bus;
// assign mem8_in = alu_c;
// assign mem16_in = alu_c;

  // Write back - destination selection
  // see wb_dest.generated.v
  always @* begin
    rf_write = 1'b0;
    write_ar = 3'b0;
        mem8_write = 1'b0;
        mem16_write = 1'b0;
        mem_we = 1'b0;
    case(dec_writeback_dest_mux_sel)
      USEQ_SEL_DEST_RFC: begin
        rf_write = 1'b1;
      end
      USEQ_SEL_DEST_COND_LINK_REG: begin
        rf_write = (dec_cond == USEQ_COND_ALAL);  // we
            write to the link register only if the
           condition field
        // is set to USEQ_COND_ALAL (ALways And Link)
      end
      USEQ_SEL_DEST_MEM16: begin
        // Fixme
                mem16_write = 1'b1;
                mem_we = 1'b1;
      end
      USEQ_SEL_DEST_MEM8: begin
                mem8_write = 1'b1;
        mem_we = 1'b1;
                // Fixme
      end
      USEQ_SEL_DEST_SR: begin
        case(dec_rd)
         USEQ_SR_AR0 : begin
           write_ar = 3'b001;
         end
         USEQ_SR_AR1 : begin
           write_ar = 3'b010;
```

```
          end
          USEQ_SR_AR2 : begin
            write_ar = 3'b100;
          end
           default: begin
             write_ar = 3'b000;
           end

        endcase
      end
      default: begin
        rf_write = 1'b0;
        write_ar = 3'b0;
      end
    endcase
  end

  // Selection of special register for read
  // To be used by mfrs instruction

  always @* begin
    case(dec_rs1)
      USEQ_SR_AR0 : begin
        sr_read_bus = ar0_r;
      end
      USEQ_SR_AR1 : begin
        sr_read_bus = ar1_r;
      end
      USEQ_SR_AR2 : begin
        sr_read_bus = ar2_r;
      end
      USEQ_SR_SR : begin
        sr_read_bus = 0; // Fixme
      end
      default: begin
      end

    endcase
  end

  // Halt instruction
```

```
always @* begin
       if(rst_n == 0) begin
               halt = 1'b0;
       end else begin
  case(dec_flow_halt_mux_sel)
    USEQ_FLOW_HALT_YES: begin
      halt = 1'b1;
    end
    USEQ_FLOW_HALT_NO: begin
      halt = 1'b0;
    end
     default: begin
      halt = 1'b0;
    end
  endcase
       end
end
```

```
wire [15:0] decode_string;

assign decode_string = {dec_opcode1,dec_opcode3,
   dec_opcode4,dec_cond};
reg [39:0]          _codeop_ascii_r;          //
   Decode of state_r
initial
  _codeop_ascii_r = "Unknown";


always @* begin
  // FIXME : we need the default case
  casez(decode_string)
  //@begin[case_inst]
  USEQ_OPCODES_AND: begin
       _codeop_ascii_r = "USEQ_OPCODES_AND";
       dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
```

```
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LDIR1: begin
```

```
        _codeop_ascii_r = "USEQ_OPCODES_LDIR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R1;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K12;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_HALT: begin
        _codeop_ascii_r = "USEQ_OPCODES_HALT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_YES;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
```

```
    end
    USEQ_OPCODES_XORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LB: begin
        _codeop_ascii_r = "USEQ_OPCODES_LB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM8;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_NOT_RFB
            ;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
```

```
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_RET: begin
        _codeop_ascii_r = "USEQ_OPCODES_RET";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_PC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R14;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_FROM_ALU;
    end
    USEQ_OPCODES_LW: begin
        _codeop_ascii_r = "USEQ_OPCODES_LW";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM16;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB3: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
```

```
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SUB;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BRPC: begin
        _codeop_ascii_r = "USEQ_OPCODES_BRPC";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RS2;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K8;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_NPC;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_SUB2: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
```

```
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_NOT_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ANDR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ANDR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MFSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MFSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_SR;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MTSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MTSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_SR;
```

```
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORI4: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORI4";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRAI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRAI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SEXT: begin
```

```
        _codeop_ascii_r = "USEQ_OPCODES_SEXT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SEXT;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BR: begin
        _codeop_ascii_r = "USEQ_OPCODES_BR";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_XOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
```

```
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRA: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRA";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD3: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
```

```
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SW: begin
        _codeop_ascii_r = "USEQ_OPCODES_SW";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_MEM16;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
```

```
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SB: begin
        _codeop_ascii_r = "USEQ_OPCODES_SB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_MEM8
            ;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
```

```
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_OR: begin
        _codeop_ascii_r = "USEQ_OPCODES_OR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
   //@end[case_inst]
   default: begin

   end
   endcase
  end


  wire cond_branch;
  reg equal_flag_r;
  wire alu_equal;
  assign alu_equal = (alu_b == alu_c);

   always@(*)  begin
   casez(decode_string)
    USEQ_OPCODES_ADD2: begin
                  equal_flag_r <= alu_equal;
        end
```

```
    endcase
  end


assign cond_branch = equal_flag_r;
reg ram_delay;
reg ram_delay_done;


  always @(posedge clk or negedge rst_n) begin
   if(rst_n == 1'b0) begin
                npc <= 0;
                ram_delay<=0;
                end
        else if(halt) begin
      npc <= pc_r;
    end else begin
        `ifdef negativeedge
        `else
        if(ram_delay==1) begin
                 ram_delay<=0;
        //       ram_delay_done = 1;
        //end else if (ram_delay_done==1) begin
                //ram_delay_done <= 0;
        end else begin
        casez(decode_string)
                USEQ_OPCODES_SW: begin
                        ram_delay<= 1;
                end
                USEQ_OPCODES_SB: begin
                        ram_delay<= 1;
                end
                USEQ_OPCODES_LB:  begin
                        ram_delay <= 1;
                end
                USEQ_OPCODES_LW: begin
                        ram_delay <=1;
                end
                default: begin
                 ram_delay<=0;
                end
```

```
                endcase
          end
          `endif
          if(ram_delay==0) begin

      case(dec_pc_next_mux_sel)
        USEQ_SEL_PC_INCREMENT: begin
            npc <= pc_r + USEQ_PC_INC;
        end
        USEQ_SEL_PC_FROM_ALU: begin
            npc <= dest_bus;
        end
        USEQ_SEL_PC_COND_LOAD_FROM_ALU: begin
          if(cond_branch) begin
           npc <= dest_bus;
          end
          else begin
            npc <= pc_r + USEQ_PC_INC;
          end
          end

        default: begin
          npc <= pc_r + USEQ_PC_INC;
        end

      endcase
          end else begin
                  npc <= pc_r;
                  end
                  end
end




  // Program counter management
/*    always @* begin
      if(halt | halt_r) begin
        npc = pc_r;
      end
```

```
    else if (write_to_pc) begin
      npc <= dest_bus;
    end
    else begin
      npc <= pc_r + USEQ_PC_INC;
    end */


  //end
 /* always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      // Beginning of autoreset for uninitialized flops
      pc_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
          ram_delay<=0;
          ram_delay_done <= 0;
      // End of automatics
    end
  else begin
        if(ram_delay==1) begin
                ram_delay<=0;
        //      ram_delay_done = 1;
        //end else if (ram_delay_done==1) begin
                //ram_delay_done <= 0;
        end else begin

        casez(decode_string)
                USEQ_OPCODES_SW: begin
                        ram_delay <= 1;
                end
                USEQ_OPCODES_SB: begin
                        ram_delay <= 1;
                end
                USEQ_OPCODES_LB:  begin
                        ram_delay <= 1;
                end
                USEQ_OPCODES_LW: begin
                        ram_delay <=1;
                end
                default: begin
                 ram_delay<=0;
                end
```

```
                endcase
        end

        if(ram_delay==0) begin
          pc_r <= npc;
        end
          halt_r <= halt;
        end
end // always @ (posedge clk or negedge rst_n)
*/


assign carry_in = 0; // FIXME

simd_useq_alu U_ALU (
                     // Outputs
                     .c                (alu_c[
                        USEQ_WORD_MSB:0]),
                     // Inputs
                     .a                (alu_a[
                        USEQ_WORD_MSB:0]),
                     .b                (alu_b[
                        USEQ_WORD_MSB:0]),
                     .op               (
                        dec_alu_op_mux_sel[
                        USEQ_SEL_ALUOP_MSB:0]),
                     .carry_in         (carry_in),
                     .carry_out        (carry_out));



 /* simd_useq_rf AUTO_TEMPLATE(
  ); */
simd_useq_rf U_RF (
                   .rf_a               (rf_a[
                      USEQ_WORD_MSB:0]),
                   .rf_b               (rf_b[
                      USEQ_WORD_MSB:0]),
                   /*AUTOINST*/
                   // Inputs
```

```
                    .rf_sela            (rf_sela[
                      USEQ_RF_SEL_MSB:0]),
                    .rf_selb            (rf_selb[
                      USEQ_RF_SEL_MSB:0]),
                    .rf_selc            (rf_selc[
                      USEQ_RF_SEL_MSB:0]),
                    .rf_c               (rf_c[
                      USEQ_WORD_MSB:0]),
                    .rf_write           (rf_write),
                    .clk                (clk),
                    .rst_n              (rst_n));



/* always @(posedge clk or negedge rst_n) begin
    if(rst_n == 1'b0) begin
      //AUTORESET
      // Beginning of autoreset for uninitialized flops
      dec_inst_r <= {(1+(USEQ_INST_MSB)){1'b0}};
      // End of automatics
    end
    else begin
      dec_inst_r <= instruction;

    end
 end
 */



simd_useq_decoder U_DC3 (.dec_opcode1          (
    dec_opcode1[USEQ_INST_FORMAT_OPCODE1_MSB:0]),

                                                  .
                                                    dec_rd



                                                  (
                                                  dec_rd
                                                  [
```

```
USEQ_INST_FO
:0])
,

.
dec_opcode3

(
dec_opcode3
[
USEQ_INST_FO
:0])
,

.
dec_rs2

(
dec_rs2
[
USEQ_INST_FO
:0])
,

.
dec_uk4

(
dec_uk4
[
USEQ_INST_FO
:0])
,
```

```
.
    dec_rs1

    (
    dec_rs1
    [
    USEQ_INST_FO
    :0])
    ,

.
    dec_opcode4

    (
    dec_opcode4
    [
    USEQ_INST_FO
    :0])
    ,

.
    dec_k8

    (
    dec_k8
    [
    USEQ_INST_FO
    :0])
    ,

.
    dec_cond

    (
```

```
dec_cond
[
USEQ_INST_FO
:0])
,

.
dec_k12


(
dec_k12
[
USEQ_INST_FC
:0])
,

.
dec_k4


(
dec_k4
[
USEQ_INST_FO
:0])
,

.
instruction

(
instruction
[
USEQ_INST_MS
:0])
,
```

```
.
    clk


    (
    clk
    )
    ,

.
    rst_n


    (
    rst_n
    )
    )
    ;
```

```
endmodule // simd_useq
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.P - Asynchronous advanced FIFO CPU core

```
module delay_cell (z,i);
        parameter DELAY = 1;
        parameter NR_BITS = $clog2(DELAY);
```

```
        input i;
        output z;
        wire [NR_BITS-1:0] delay_sel;
        assign delay_sel = DELAY-1;
        delay_module #(.DELAY(DELAY)) DELMOD(.out(z),.
            in(i),.sel(delay_sel[NR_BITS-1:0]));
        //assign #DELAY z = i;
endmodule

module delay_module(out,in,sel) ;
parameter DELAY = 1;
parameter NR_BITS = $clog2(DELAY);
        input in;
        input [NR_BITS-1:0] sel;
        output out;
        reg out;
        //ORDLY15s8fhvt DELAYEL(out,in,1'b1);
        wire [DELAY-1:0] in_temp;
        wire test;

        genvar i;
                ORDLY15s8fhvt DELAYEL(.o(in_temp[0]),.
                    i1(in),.i2(1'b0));
        generate
                for(i=0;i<DELAY-1; i=i+1) begin:
                    delaygen
                                ORDLY15s8fhvt DELAYEL(.
                                    o(in_temp[i+1]),.i1(
                                    in_temp[i]) ,.i2(1'
                                    b0));
                end
        endgenerate

        always @ * begin
                out      <= in_temp[sel];
        end


        endmodule
module two_to_one_mux_nand(in1, in2, sel, out)    ;
        input in1,in2,sel;
```

```
        output out;

        wire a,b,c;
        not n1(a,sel);
        nand a1(b,in1,a);

        nand n2(c,in2,sel);

        nand n3(out,c,b);
endmodule


module muller_c_element(a,b,reset,c);

        input a,b,reset;
        output c;
        cmuller2d4s8fhvt MULL ( .a(a), .b(b), .rstneg(
           reset), .z(c) );
        /* wire d,e,f,g;
        nand(d,a,b);
        nand(e,a,c);
        nand(f,b,c);
        nand #2 (g,d,e,f);
        //added mux to reset
        two_to_one_mux_nand mux1(0, g, reset, c); */
endmodule
/*
module pipe_module(x_req,y_ack,reset,data_in,data_out,
   x_ack,y_req);
parameter DATASIZE=16;
parameter DELAY=2;
        input x_req,y_ack,reset;
        input [DATASIZE-1:0] data_in;
        output x_ack,y_req;
        output [DATASIZE-1:0] data_out;
        wire not_y_ack;
        wire c;
        assign not_y_ack = !y_ack;
        assign x_ack = c;
        muller_c_element M1(x_req,not_y_ack,reset,c);
```

```
        ev_ctrl_storage_byte #(.SIZE(DATASIZE)) SE (
           data_in,c,y_ack,data_out);
        delay_cell #(.DELAY(DELAY)) DC1 (c,y_req);
endmodule
*/
//Register elementdec_rf_sela_mux_sel
module ev_ctrl_storage_element(in,c,p,out);      //The
   fast one
        input in,c,p;
        output out;
        wire a,b,d,e,f,g,h;
reg out;
        always @* begin
                if(c==0 && p== 0) begin
                        out<= in;
                       end else if(c==1 && p==0) begin
                       out<= out;
                        end else if (c==1 && p==1)
                           begin
                         out<= in;
                         end else if (c==0 && p==1)
                            begin
                         out<= out;
                         end
                         end
        ////first top mux
        // two_to_one_mux_nand U1(a,in,c,b);
        // not(d,b);
        // not(a,d);

        ////bottom mux
        // two_to_one_mux_nand U2(in,e,c,f);
        // not(g,f);
        // not(e,g);

        ////out mux
        // two_to_one_mux_nand U3(g,d,p,h);
        // not (out,h);
endmodule

module ev_ctrl_storage_byte(in,c,p,out);
```

```
        parameter SIZE=8;

        input [SIZE-1:0] in;
        input c,p;
        output [SIZE-1:0]out;

        genvar i;
        ev_ctrl_storage_element REGEL(in[0],c,p,out[0])
           ;
        generate
                for(i=0;i<SIZE; i=i+1) begin:reggen
                ev_ctrl_storage_element REGEL(in[i],c,p
                   ,out[i]);
                end
        endgenerate


endmodule

module toggle_on_change(out,a,b,reset);
        input a,b,reset;
        output out;
        reg out;

        always @(a or b or  reset) begin
                if (reset==0) begin
                        out <= 0;
                end else begin
                        out <= !out;
                end
        end
endmodule

module pulse_generator(out,in);
        input in;
        output out;
        wire in;
        wire out;
        wire a;
        delay_cell  #(.DELAY (1)) D1(.z(a),.i(in));
```

```
        xor2d3s8fhvt PXOR(.I1(a), .I2(in), .O(out));
endmodule



module simd_useq(/*AUTOARG*/
  // Outputs
  pc_r, mem_in, mem_addr, mem_cs,mem_we,mem_clk,
     cpu_ack, ram_clk,
  // Inputs
  instruction, cpu_req, rst_n,mem_out
  );

`include "simd_params.v"

  input  [USEQ_INST_MSB:0] instruction; // Instruction
     word fetched from memory
  output [USEQ_WORD_MSB:0] pc_r; // address for the
     code memory


  input                      rst_n;
  input                     cpu_req;
   input                           [USEQ_WORD_MSB:0]
            mem_out;
  output                  cpu_ack;
  output                                          ram_clk
     ;

     output mem_cs,mem_clk;
  output mem_we;
  output  [USEQ_WORD_MSB:0] mem_addr;
  output [USEQ_WORD_MSB:0] mem_in;


  /*AUTOINPUT*/
  /*AUTOOUTPUT*/

  /*AUTOREG*/
  /*AUTOWIRE*/
```

```
wire [USEQ_WORD_MSB:0]     dec_inst_r;
wire [USEQ_WORD_MSB:0]                dec_inst_rr;


// Register file
reg [3:0]                      rf_sela;
reg [3:0]                      rf_selb;
reg [3:0]                      rf_selc;
reg                            rf_write;

wire [USEQ_WORD_MSB:0]         rf_a_out;
wire [USEQ_WORD_MSB:0]         rf_b_out;

reg [USEQ_WORD_MSB:0]         rf_a;
reg [USEQ_WORD_MSB:0]         rf_b;
wire [USEQ_WORD_MSB:0]         rf_c;

// alu
reg  [USEQ_WORD_MSB:0]          alu_a;
reg  [USEQ_WORD_MSB:0]          alu_b;
wire [USEQ_WORD_MSB:0]          alu_c;

reg [USEQ_WORD_MSB:0]           dest_bus;
reg [USEQ_WORD_MSB:0]           dest_bus_s0;
reg [USEQ_WORD_MSB:0]           dest_bus_s1;
reg [USEQ_WORD_MSB:0]           dest_bus_s2;


// PC
reg [USEQ_WORD_MSB:0]           pc_r;
reg [USEQ_WORD_MSB:0]           npc;
reg                             write_to_pc;

// Flow control
reg                             halt;
reg                             halt_r;
```

```
  // Special register support
  reg [USEQ_WORD_MSB:0]              sr_read_bus;


  // PE array address register
  // Fixme - those must be moved outside uSeq, to the
     address generator block
  // We keep them here for now for instruction &
     toolchain tests

  reg [USEQ_WORD_MSB:0]              ar0_r;
  reg [USEQ_WORD_MSB:0]              ar1_r;
  reg [USEQ_WORD_MSB:0]              ar2_r;
  reg [2:0]                          write_ar;


//@begin[sig_mux_sel]
reg [USEQ_SEL_RFB_MSB:0]  dec_rf_selb_mux_sel;
reg [USEQ_SEL_ALUB_MSB:0]  dec_alu_sourceb_mux_sel;
reg [USEQ_SEL_DEST_MSB:0]  dec_writeback_dest_mux_sel;
reg [USEQ_SEL_PC_MSB:0]  dec_pc_next_mux_sel;
reg [USEQ_SEL_RFC_MSB:0]  dec_rf_selc_mux_sel;
reg [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel;
reg [USEQ_SEL_ALUOP_MSB:0]  dec_alu_op_mux_sel;
reg [USEQ_SEL_ALUA_MSB:0]  dec_alu_sourcea_mux_sel;
reg [USEQ_SEL_RFA_MSB:0]  dec_rf_sela_mux_sel;
reg [USEQ_FLOW_HALT_MSB:0]  dec_flow_halt_mux_sel;
//@end[sig_mux_sel]

   wire mem_cs;
   reg mem_clk;
  reg mem_we;
  reg  [USEQ_WORD_MSB:0] mem_addr;
  wire [USEQ_WORD_MSB:0] mem_in;

wire [USEQ_WORD_MSB:0] mem16_out;
reg [USEQ_WORD_MSB:0] mem16_in ;
reg mem16_write;
reg [USEQ_WORD_MSB:0] mem16_addr ;
```

```
wire [USEQ_BYTE_MSB:0] mem8_out;
reg [USEQ_BYTE_MSB:0] mem8_in;
reg mem8_write;
reg [USEQ_WORD_MSB:0] mem8_addr;

reg mem_req;

                wire ram_clk;
      reg rf_clk;


 reg alu_req;
 wire alu_ack;

  wire rf_done;
     wire rf_ack;
          reg ram_req;
          reg ram_req_stall;
          reg ram_req_pre;
  wire ram_ack;
  reg rf_req;
  wire cpu_ack;
    wire mem_clk_pre;
   wire pc_r_clk;
    wire mem_req_delayed;
reg  rf_req_pre;
wire mem_req_pre;
reg ram_done;
wire ram_done_pre;
wire alu_done;
wire mem_done;
wire mem_ack;


   wire [15:0] decode_string;
    reg [15:0] FIFO_pc_r0;
        reg [15:0] FIFO_pc_r1;
        reg [15:0] FIFO_pc_r2;
        reg [15:0] FIFO_pc_r3;
         reg [3:0] FIFO_rd0;
         reg [3:0] FIFO_rd1;
```

```
          reg [3:0] FIFO_rd2;
          reg [3:0] FIFO_rd3;
           reg [15:0] FIFO_op0;
           reg [15:0] FIFO_op1;
           reg [15:0] FIFO_op2;
           reg [15:0] FIFO_op3;
           wire ram_done_clk;
           wire ram_ack_clk;

   reg [2:0] stall_cnt;
reg [2:0] stall_target;
reg [15:0] alu_c_stored;
reg alu_rf_b_forward;
reg alu_rf_a_forward;
reg stalling;
 reg stall;
 wire alu_done_clk;
 wire rf_clk_pre;

 assign alu_done = mem_req_pre;
  //@begin[inst_decode]
wire [USEQ_INST_FORMAT_OPCODE1_MSB:0] dec_opcode1 =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE1_OFFSET +:
   USEQ_INST_FORMAT_OPCODE1_SIZE];

wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd = dec_inst_r[
   USEQ_INST_FORMAT_RD_OFFSET +:
   USEQ_INST_FORMAT_RD_SIZE];

wire [USEQ_INST_FORMAT_OPCODE3_MSB:0] dec_opcode3 =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE3_OFFSET +:
   USEQ_INST_FORMAT_OPCODE3_SIZE];

wire [USEQ_INST_FORMAT_RS2_MSB:0] dec_rs2 = dec_inst_r
   [USEQ_INST_FORMAT_RS2_OFFSET +:
   USEQ_INST_FORMAT_RS2_SIZE];

wire [USEQ_INST_FORMAT_UK4_MSB:0] dec_uk4 = dec_inst_r
   [USEQ_INST_FORMAT_UK4_OFFSET +:
   USEQ_INST_FORMAT_UK4_SIZE];
```

```
wire [USEQ_INST_FORMAT_RS1_MSB:0] dec_rs1 = dec_inst_r
   [USEQ_INST_FORMAT_RS1_OFFSET +:
   USEQ_INST_FORMAT_RS1_SIZE];

wire [USEQ_INST_FORMAT_OPCODE4_MSB:0] dec_opcode4 =
   dec_inst_r[USEQ_INST_FORMAT_OPCODE4_OFFSET +:
   USEQ_INST_FORMAT_OPCODE4_SIZE];

wire [USEQ_INST_FORMAT_K8_MSB:0] dec_k8 = dec_inst_r[
   USEQ_INST_FORMAT_K8_OFFSET +:
   USEQ_INST_FORMAT_K8_SIZE];

wire [USEQ_INST_FORMAT_COND_MSB:0] dec_cond =
   dec_inst_r[USEQ_INST_FORMAT_COND_OFFSET +:
   USEQ_INST_FORMAT_COND_SIZE];

wire [USEQ_INST_FORMAT_K12_MSB:0] dec_k12 = dec_inst_r
   [USEQ_INST_FORMAT_K12_OFFSET +:
   USEQ_INST_FORMAT_K12_SIZE];

wire [USEQ_INST_FORMAT_K4_MSB:0] dec_k4 = dec_inst_r[
   USEQ_INST_FORMAT_K4_OFFSET +:
   USEQ_INST_FORMAT_K4_SIZE];

  //@end[inst_decode]


//PULSE ELEMENT FOR THIS
  // temporay code for address registers
  always @(rf_done or rst_n) begin
    if(rst_n == 1'b0) begin
      /*AUTORESET*/
      // Beginning of autoreset for uninitialized flops
      ar0_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar1_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      ar2_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
      // End of automatics
    end
    else begin
      if(write_ar[2:0] == 3'b001)
        ar0_r <= dest_bus;
```

```
        if(write_ar[2:0] == 3'b010)
          ar1_r <= dest_bus;
        if(write_ar[2:0] == 3'b100)
          ar2_r <= dest_bus;
      end
    end

assign mem_done = rf_req;



//wires for latches
//s0
wire [USEQ_WORD_MSB:0] alu_a_s0;
wire [USEQ_WORD_MSB:0] alu_b_s0;
wire [USEQ_SEL_ALUOP_MSB:0] dec_alu_op_mux_sel_s0;
wire [3:0] rf_selc_s0;
wire rf_write_s0;
wire [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel_s0;
wire [USEQ_WORD_MSB:0] rf_b_s0;
wire [USEQ_SEL_PC_MSB:0] dec_pc_next_mux_sel_s0;
wire [USEQ_SEL_RFC_MSB:0] dec_rf_selc_mux_sel_s0;
wire [15:0] decode_string_s0;
wire [USEQ_SEL_DEST_MSB:0]
   dec_writeback_dest_mux_sel_s0;
wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd_s0;
wire halt_s0;

//s1
wire [USEQ_WORD_MSB:0] alu_c_s1;
wire rf_write_s1;
wire [3:0] rf_selc_s1;
wire [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel_s1;
wire [USEQ_WORD_MSB:0] rf_b_s1;
wire [USEQ_SEL_PC_MSB:0] dec_pc_next_mux_sel_s1;
wire [USEQ_SEL_RFC_MSB:0] dec_rf_selc_mux_sel_s1;
wire [15:0] decode_string_s1;
wire [USEQ_SEL_DEST_MSB:0]
   dec_writeback_dest_mux_sel_s1;
```

```
wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd_s1;

//s2
wire [USEQ_WORD_MSB:0] alu_c_s2;
wire rf_write_s2;
wire [3:0] rf_selc_s2;
wire [USEQ_SEL_WBSOURCE_MSB:0]
   dec_writeback_source_mux_sel_s2;
wire [USEQ_BYTE_MSB:0] mem8_out_s2;
wire [USEQ_WORD_MSB:0] mem16_out_s2;
wire [USEQ_SEL_RFC_MSB:0] dec_rf_selc_mux_sel_s2;
wire [USEQ_SEL_DEST_MSB:0]
   dec_writeback_dest_mux_sel_s2;
wire [USEQ_WORD_MSB:0] rf_b_s2;
wire [USEQ_INST_FORMAT_RD_MSB:0] dec_rd_s2;

always @* begin
if(rst_n == 1'b0) begin
alu_req = 1'b0;

end else if (stall == 1) begin
        alu_req = ram_req_stall;
end else begin
        alu_req = ram_done;
end
end
//assign alu_req = ram_done;

assign mem_in = alu_c;
assign mem_cs = 0;
assign mem16_out = mem_out;
assign mem8_out = mem_out;

always @* begin
if(rst_n == 1'b0) begin
ram_req = 1'b0;
ram_req_stall = 1'b0;
end else if(stall == 1'b1) begin
        ram_req_stall = ram_req_pre;
        ram_req = ram_req;
        end else begin
```

```
        ram_req = ram_req_pre;
        ram_req_stall = ram_req_stall;
        end
end

always @*begin
if(rst_n == 1'b0) begin
ram_done = 0;
end else if(stall==1) begin
ram_done = ram_req_stall;
end else begin
ram_done = ram_done_pre;
end
end

muller_c_element RAMC (ram_req,!alu_ack,rst_n,ram_ack);
delay_cell #(.DELAY (10)) RAMD (.z(ram_done_pre),.i(
   ram_ack));
pulse_generator PGPCR(.out(pc_r_clk),.in(alu_ack));

muller_c_element ALUC (.a(alu_req),.b(!mem_ack),.reset(
   rst_n),.c(alu_ack));
//ev_ctrl_storage_byte    #(.SIZE (1)) SBHALT(.in(halt)
   ,.c(alu_ack),.p(mem_ack),.out(halt_s0));
ev_ctrl_storage_byte    #(.SIZE (16)) DECS(.in(
   dec_inst_rr),.c(alu_ack),.p(mem_ack),.out(dec_inst_r
   )); //can skip this latch? should latch over
   register read to?
ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
   SBALUAS0(.in(alu_a),.c(alu_ack),.p(mem_ack),.out(
   alu_a_s0));
ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
   SBALUBS0(.in(alu_b),.c(alu_ack),.p(mem_ack),.out(
   alu_b_s0));
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_ALUOP_BITS))
   SBOPMUXS0(.in(dec_alu_op_mux_sel),.c(alu_ack),.p(
   mem_ack),.out(dec_alu_op_mux_sel_s0));
//ev_ctrl_storage_byte    #(.SIZE (16)) DECS(dest_bus,.c
   (alu_ack,.p(mem_ack,dest_bus_s0); //rf_b on mem
ev_ctrl_storage_byte    #(.SIZE (4)) SBSELCS0(.in(
   rf_selc),.c(alu_ack),.p(mem_ack),.out(rf_selc_s0));
```

```
ev_ctrl_storage_byte    #(.SIZE (1)) SBRFWRS0(.in(
    rf_write),.c(alu_ack),.p(mem_ack),.out(rf_write_s0))
    ;
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_WBSOURCE_BITS)
    ) SBWBMUXS0(.in(dec_writeback_source_mux_sel),.c(
    alu_ack),.p(mem_ack),.out(
    dec_writeback_source_mux_sel_s0));
ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
    SBRFBS0(.in(rf_b),.c(alu_ack),.p(mem_ack),.out(
    rf_b_s0)); //rf_b on mem
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_PC_BITS))
    SBPCMUXS0(.in(dec_pc_next_mux_sel),.c(alu_ack),.p(
    mem_ack),.out(dec_pc_next_mux_sel_s0));
ev_ctrl_storage_byte    #(.SIZE (16)) SBDECSTRS0(.in(
    decode_string),.c(alu_ack),.p(mem_ack),.out(
    decode_string_s0));
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_DEST_BITS))
    SBWBDESTMUXS0(.in(dec_writeback_dest_mux_sel),.c(
    alu_ack),.p(mem_ack),.out(
    dec_writeback_dest_mux_sel_s0));
ev_ctrl_storage_byte    #(.SIZE (
    USEQ_INST_FORMAT_RD_SIZE)) SBDECRDS0(.in(dec_rd),.c(
    alu_ack),.p(mem_ack),.out(dec_rd_s0));
//ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_RFC_BITS))
    SBSELCMUXS0(dec_rf_selc_mux_sel,alu_ack,mem_ack,
    dec_rf_selc_mux_sel_s0);
//

delay_cell #(.DELAY (2))  ALUD (.z(mem_req_pre),.i(
    alu_ack));

 //MEMORY request  //store alu_c, dest_bus_s0,
     rf_write_s0, rf_selc_s0
 muller_c_element MEMC (.a(mem_req_pre),.b(!rf_ack),.
     reset(rst_n),.c(mem_ack));
 ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
     SBALUCS1(.in(alu_c),.c(mem_ack),.p(rf_ack),.out(
     alu_c_s1));
 //ev_ctrl_storage_byte    #(.SIZE (16)) DECS(
     dest_bus_s0),.c(mem_ack),.p(rf_ack),dest_bus_s1);
```

```
ev_ctrl_storage_byte    #(.SIZE (1)) SBRFWRS1(.in(
    rf_write_s0),.c(mem_ack),.p(rf_ack),.out(
    rf_write_s1));
ev_ctrl_storage_byte    #(.SIZE (4)) SBSELCS1(.in(
    rf_selc_s0),.c(mem_ack),.p(rf_ack),.out(rf_selc_s1)
    );
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_WBSOURCE_BITS
    )) SBWBMUXS1(.in(dec_writeback_source_mux_sel_s0),.
    c(mem_ack),.p(rf_ack),.out(
    dec_writeback_source_mux_sel_s1));
ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
    SBRFBS1(.in(rf_b_s0),.c(mem_ack),.p(rf_ack),.out(
    rf_b_s1));
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_PC_BITS))
    SBPCMUXS1(.in(dec_pc_next_mux_sel_s0),.c(mem_ack),.
    p(rf_ack),.out(dec_pc_next_mux_sel_s1));
ev_ctrl_storage_byte    #(.SIZE (16)) SBDECSTRS1(.in(
    decode_string_s0),.c(mem_ack),.p(rf_ack),.out(
    decode_string_s1));
ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_DEST_BITS))
    SBWBDESTMUXS1(.in(dec_writeback_dest_mux_sel_s0),.c
    (mem_ack),.p(rf_ack),.out(
    dec_writeback_dest_mux_sel_s1));
ev_ctrl_storage_byte    #(.SIZE (
    USEQ_INST_FORMAT_RD_SIZE)) SBDECRDS1(.in(dec_rd_s0)
    ,.c(mem_ack),.p(rf_ack),.out(dec_rd_s1));
//ev_ctrl_storage_byte    #(.SIZE (USEQ_SEL_RFC_BITS))
    SBSELCMUXS1(.in(dec_rf_selc_mux_sel_s0 ,.c(mem_ack,.
    p(rf_ack,dec_rf_selc_mux_sel_s1);


 //RF request and delay

  muller_c_element RFC (.a(rf_req),.b(!rf_done),.reset
     (rst_n),.c(rf_ack));
  ev_ctrl_storage_byte    #(.SIZE (4)) SBSELCS2(.in(
     rf_selc_s1),.c(rf_ack),.p(rf_done),.out(
     rf_selc_s2));
   ev_ctrl_storage_byte    #(.SIZE (
      USEQ_SEL_WBSOURCE_BITS)) SBWBMUXS2(.in(
      dec_writeback_source_mux_sel_s1),.c(rf_ack),.p(
```

```
        rf_done),.out(dec_writeback_source_mux_sel_s2));
         ev_ctrl_storage_byte    #(.SIZE (1)) SBRFWRS2(.
            in(rf_write_s1),.c(rf_ack),.p(rf_done),.out(
            rf_write_s2));
 ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
    SBALUCS2(.in(alu_c_s1),.c(rf_ack),.p(rf_done),.out(
    alu_c_s2));

         ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS)
            ) SBMEM8OS2(.in(mem8_out),.c(rf_ack),.p(
            rf_done),.out(mem8_out_s2));
         ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS)
            ) SBMEM16OS2(.in(mem16_out),.c(rf_ack),.p(
            rf_done),.out(mem16_out_s2));
         ev_ctrl_storage_byte    #(.SIZE (
            USEQ_SEL_DEST_BITS)) SBWBDESTMUXS2(.in(
            dec_writeback_dest_mux_sel_s1),.c(rf_ack),.p
            (rf_done),.out(dec_writeback_dest_mux_sel_s2
            ));
 ev_ctrl_storage_byte    #(.SIZE (USEQ_WORD_BITS))
    SBRFBS2(.in(rf_b_s1),.c(rf_ack),.p(rf_done),.out(
    rf_b_s2));
 ev_ctrl_storage_byte    #(.SIZE (
    USEQ_INST_FORMAT_RD_SIZE)) SBDECRDS2(.in(dec_rd_s1)
    ,.c(rf_ack),.p(rf_done),.out(dec_rd_s2));
        // ev_ctrl_storage_byte    #(.SIZE (
            USEQ_SEL_RFC_BITS)) SBSELCMUXS2(
            dec_rf_selc_mux_sel_s1,mem_ack,rf_ack,
            dec_rf_selc_mux_sel_s2);

delay_cell #(.DELAY (3))  RFD (.z(rf_done),.i(rf_ack));
      pulse_generator PGMEM(.out(mem_clk_pre),.in(
         mem_req));
  delay_cell #(.DELAY (10)) MEMD (.z(mem_req_delayed),.
     i(mem_req));
  pulse_generator PGRF(.out(rf_clk_pre),.in(rf_ack));

  always @* begin
  if(stall ==1) begin
        rf_clk <= 0;
        end else begin
```

```
        rf_clk <= rf_clk_pre;
        end
        end

        reg ram_ack_muxed;
        always @* begin
        if(rst_n== 1'b0) begin
                ram_ack_muxed = 0;
                end else if (stall== 1'b1) begin
                ram_ack_muxed = ram_req_stall;
                end else begin
                ram_ack_muxed = ram_ack;
                end
        end


  //stalling
  pulse_generator PGRAMACK(.out(ram_ack_clk),.in(
     ram_ack_muxed));
  pulse_generator PGRAMDONE(.out(ram_done_clk),.in(
     ram_done));
  pulse_generator PGALUDONE(.out(alu_done_clk),.in(
     alu_done));
  //end of pipeline

  always @(posedge alu_done_clk) begin
        if(rst_n == 1'b0) begin
                alu_c_stored <= 0;
        end else begin
                alu_c_stored <= alu_c;
        end
end

 //FIFO FOR INSTR. AND RD
 //ON RAMACK -> SHIFT FIFO AND ADD ELEMENT AT START
 always @(posedge ram_ack_clk or negedge rst_n) begin
        if(rst_n == 1'b0) begin
        FIFO_pc_r3<= {(1+(USEQ_WORD_MSB)){1'b0}};
        FIFO_pc_r2<= {(1+(USEQ_WORD_MSB)){1'b0}};
        FIFO_pc_r1<= {(1+(USEQ_WORD_MSB)){1'b0}};
```

```
        FIFO_rd3<= {(1+(USEQ_WORD_MSB))}{1'b0}};
        FIFO_rd2<= {(1+(USEQ_WORD_MSB))}{1'b0}};
        FIFO_rd1<= {(1+(USEQ_WORD_MSB))}{1'b0}};

        FIFO_op3<= {(1+(USEQ_WORD_MSB))}{1'b0}};
        FIFO_op2<= {(1+(USEQ_WORD_MSB))}{1'b0}};
        FIFO_op1<= {(1+(USEQ_WORD_MSB))}{1'b0}};
        end else begin
        FIFO_pc_r3<= FIFO_pc_r2;
        FIFO_pc_r2<= FIFO_pc_r1;
        FIFO_pc_r1<= FIFO_pc_r0;

        FIFO_rd3<= FIFO_rd2;
        FIFO_rd2<= FIFO_rd1;
        FIFO_rd1<= FIFO_rd0;

        FIFO_op3<= FIFO_op2;
        FIFO_op2<= FIFO_op1;
        FIFO_op1<= FIFO_op0;
        end
end

 //ON RAM_DONE_CLK
 always @(posedge ram_done_clk or negedge rst_n) begin
 if(rst_n == 1'b0) begin
        FIFO_pc_r0 ={(1+(USEQ_WORD_MSB))}{1'b0}};
        FIFO_rd0 ={(4){1'b0}};
        FIFO_op0 ={(1+(USEQ_WORD_MSB))}{1'b0}};
        alu_rf_b_forward =0;
        alu_rf_a_forward = 0;
        stall=0;
        stall_target <=0;
end else begin
        FIFO_pc_r0 =pc_r;
        FIFO_rd0 =rf_selc;
        FIFO_op0 =decode_string;
        if(((FIFO_rd1==rf_selb) || (FIFO_rd1==rf_sela))
            && (FIFO_pc_r1!=pc_r)) begin
        //SEND BACK ALU_C UNLESS FIFO_op == LW or SW
           then stall long WCDELAY-RAM_DELAY
                casez(FIFO_op1)
```

```
USEQ_OPCODES_SW: begin
        stall_target <=3;
        stall=1;
        alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
end
USEQ_OPCODES_SB: begin
        stall_target <=3;
        stall=1;
        alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
end
USEQ_OPCODES_LB:  begin
        stall_target <=3;
        stall=1;
        alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
end
USEQ_OPCODES_LW: begin
        stall_target <=3;
        stall=1;
        alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
end
default: begin
 stall=0;
 stall_target <=0;
 if(FIFO_rd1 == rf_sela) begin
        alu_rf_b_forward = 1;
end else begin
        alu_rf_b_forward = 0;
        end
if(FIFO_rd1 == rf_selb) begin
        alu_rf_a_forward = 1;
                end else begin
        alu_rf_b_forward = 0;
        end

end
endcase
```

```
        end else if((FIFO_rd2==rf_selb) && (FIFO_pc_r2
           !=pc_r)) begin
        alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
                                  casez(FIFO_op2)
                USEQ_OPCODES_SW: begin
                        stall_target <=2;
                        stall=1;
                end
                USEQ_OPCODES_SB: begin
                        stall_target <=2;
                        stall=1;
                end
                USEQ_OPCODES_LB:  begin
                        stall_target <=2;
                        stall=1;
                end
                USEQ_OPCODES_LW: begin
                        stall_target <=2;
                        stall=1;
                end
                default: begin
                  stall=0;
                  stall_target <=0;
                end
                endcase
//if FIFO_OP STALL till register is updated  stall
   WCDELAY-RAM_DELAY-ALU_DELAY

        end else if((FIFO_rd3==rf_selb) && (FIFO_pc_r2
           !=pc_r)) begin
                alu_rf_a_forward = 0;
        alu_rf_b_forward = 0;
//if FIFO_OP STALL till register is updated stall RF
   DELAY
                casez(FIFO_op3)
                USEQ_OPCODES_SW: begin
                        stall_target <=1;
                        stall=1;
                end
```

```
                USEQ_OPCODES_SB: begin
                        stall_target <=1;
                        stall=1;
                end
                USEQ_OPCODES_LB:  begin
                        stall_target <=1;
                        stall=1;
                end
                USEQ_OPCODES_LW: begin
                        stall_target <=1;
                        stall=1;
                end
                default: begin
                 stall=0;
                 stall_target <=0;
                 end
                endcase
        end     else begin
                if(ram_req == ram_req_stall) begin
                        stall=0;
                end else begin
                        stall = 0;
                end
                alu_rf_a_forward = 0;
                alu_rf_b_forward = 0;
                end


        end
end

        assign dec_inst_rr = instruction;


  // decoder for register file, selection for port A
  // Constants in the file rf_sela.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_sela_mux_sel)
      USEQ_SEL_RFA_R0: begin
        rf_sela <= 0;
      end
```

```
      USEQ_SEL_RFA_RS1: begin
        rf_sela <= dec_rs1;
      end
      USEQ_SEL_RFA_RD: begin
        rf_sela <= dec_rd;
      end
      default:
        rf_sela <= dec_rs1;
    endcase
  end




  // decoder for register file, selection for port B
  // Constants in the file rf_selb.generated.v (and
     useq_params.v of course)

  always @* begin
    case(dec_rf_selb_mux_sel)
          USEQ_SEL_RFB_RD: begin
                rf_selb <= dec_rd;
          end
      USEQ_SEL_RFB_R1: begin
        rf_selb <= 'b1;
      end
      USEQ_SEL_RFB_RS1: begin
        rf_selb <= dec_rs1;
      end
      USEQ_SEL_RFB_RS2: begin
        rf_selb <= dec_rs2;
      end
      default:
        rf_selb <= dec_rs2;
    endcase
  end

  always @* begin
    case(alu_rf_b_forward)
          0: begin
                rf_b <= rf_b_out;
```

```
          end
      1: begin
        rf_b <= alu_c_stored;
         end
      default:
        rf_b <= rf_b_out;
    endcase
  end


  always @* begin
    case(alu_rf_a_forward)
          0: begin
                rf_a <= rf_a_out;
          end
      1: begin
        rf_a <= alu_c_stored;
         end
      default:
        rf_a <= rf_a_out;
    endcase
  end


  // decoder for register file, selection for port C (
     writeback)
  // Constants in the file rf_selc.generated.v (and
     useq_params.v of course)
  always @* begin
    case(dec_rf_selc_mux_sel)
      USEQ_SEL_RFC_R0: begin
        rf_selc <= 0; // Fixme : really needed ?
      end
      USEQ_SEL_RFC_R1: begin
        rf_selc <= 'b1;
      end
      USEQ_SEL_RFC_RS2: begin
        rf_selc <= dec_rs2;
      end
      USEQ_SEL_RFC_RD: begin
        rf_selc <= dec_rd;
```

```
      end
      default:
        rf_selc <= dec_rd;
    endcase
  end


  // decoder for ALU port A
  // see alu_sourcea.generated.v
  always @* begin
    case(dec_alu_sourcea_mux_sel)
      USEQ_SEL_ALUA_RFA: begin
        alu_a <= rf_a;
      end
      USEQ_SEL_ALUA_NPC: begin
        alu_a <= npc;
      end
    default:
      alu_a <= rf_a;
    endcase
  end


  event evt_dbg;

  // decoder for ALU port B
  // see alu_sourcea.generated.v
  always @* begin
    case(dec_alu_sourceb_mux_sel)
      USEQ_SEL_ALUB_RFB: begin
        alu_b <= rf_b;
      end
      USEQ_SEL_ALUB_NOT_RFB: begin
        alu_b <= ~rf_b;
      end
      USEQ_SEL_ALUB_UK4: begin
        alu_b <= {{(USEQ_WORD_MINUS_K4){1'b0}},dec_uk4
            };
      end
      USEQ_SEL_ALUB_K4: begin // Sign extention
```

```
        alu_b <= {{(USEQ_WORD_MINUS_K4+1){dec_uk4[3]}},
            dec_uk4[2:0]};
      end
      USEQ_SEL_ALUB_K8: begin // Sign extention
        alu_b <= {{(USEQ_WORD_MINUS_K8+1){dec_k8[7]}},
            dec_k8[6:0]};
      end
      USEQ_SEL_ALUB_K12: begin // no sign extention
        alu_b <= {dec_k12,{(USEQ_WORD_MINUS_K12){1'b0
            }}};
      end

      USEQ_SEL_ALUB_RFB_OR_UK4: begin
        alu_b <= rf_b | {{(USEQ_WORD_MINUS_K4){1'b0}},
            dec_uk4};
      end
      USEQ_SEL_ALUB_NOT_RFB_OR_UK4: begin
        alu_b <= ~(rf_b | {{(USEQ_WORD_MINUS_K4){1'b0
            }},dec_uk4});
      end
      USEQ_SEL_ALUB_SR: begin
        alu_b <= sr_read_bus;
      end

      default:
        alu_b <= rf_b;
    endcase
  end


  // Write back - source selection
  // see wb_source.generated.v
  always @* begin
    case(dec_writeback_source_mux_sel)
      USEQ_SEL_WBSOURCE_ALU: begin
        dest_bus <= 0 ;
      end
      USEQ_SEL_WBSOURCE_RFB: begin
        dest_bus <= rf_b;
      end
      USEQ_SEL_WBSOURCE_MEM16: begin
```

```
        dest_bus <= 0;
      end
      USEQ_SEL_WBSOURCE_MEM8: begin
        dest_bus <= 0;
      end
     default:
       dest_bus <= 0;
   endcase
end

 always @* begin
  case(dec_writeback_source_mux_sel_s0)
    USEQ_SEL_WBSOURCE_ALU: begin
      dest_bus_s0 <= 0;
    end
    USEQ_SEL_WBSOURCE_RFB: begin
      dest_bus_s0 <= rf_b_s0;
    end
    USEQ_SEL_WBSOURCE_MEM16: begin
      dest_bus_s0 <= 0;
    end
    USEQ_SEL_WBSOURCE_MEM8: begin
      dest_bus_s0 <= 0;
    end
   default:
     dest_bus <= 0;
  endcase
end

always @* begin
  case(dec_writeback_source_mux_sel_s1)
    USEQ_SEL_WBSOURCE_ALU: begin
      dest_bus_s1 <= alu_c_s1;
              mem16_addr <=rf_b_s1;
              mem8_addr <= rf_b_s1;
              mem_addr <= rf_b_s1;
    end
    USEQ_SEL_WBSOURCE_RFB: begin
      dest_bus_s1 <= rf_b_s1;
              mem16_addr <=rf_b_s1;
              mem8_addr <= rf_b_s1;
```

```
                mem_addr <= rf_b_s1;
                mem16_in <= alu_c_s1;
                mem8_in <= alu_c_s1;
      end
      USEQ_SEL_WBSOURCE_MEM16: begin
        dest_bus_s1 <= mem_out;
                mem16_addr <=alu_c_s1;
                mem8_addr <=alu_c_s1;
                mem16_in <= alu_c_s1;
                mem_addr <= alu_c_s1;
      end
      USEQ_SEL_WBSOURCE_MEM8: begin
        dest_bus_s1 <= mem_out;
                mem16_addr <=alu_c_s1;
                mem8_addr <=alu_c_s1;
                mem8_in <= alu_c_s1;
                mem_addr <= alu_c_s1;
      end
     default: begin
       dest_bus_s1 <= alu_c_s1;
                mem8_addr <=rf_b_s1;
                mem8_in <= alu_c_s1;
                mem16_addr <=rf_b_s1;
                mem16_in <= alu_c_s1;
                mem_addr <= rf_b_s1;
                end
   endcase
end

    always @* begin
   case(dec_writeback_source_mux_sel_s2)
      USEQ_SEL_WBSOURCE_ALU: begin
        dest_bus_s2 <= alu_c_s2;
      end
      USEQ_SEL_WBSOURCE_RFB: begin
        dest_bus_s2 <= rf_b_s2;
      end
      USEQ_SEL_WBSOURCE_MEM16: begin
        dest_bus_s2 <= mem16_out_s2;
      end
      USEQ_SEL_WBSOURCE_MEM8: begin
```

```
        dest_bus_s2 <= mem8_out_s2;
      end
     default:
       dest_bus_s2 <= alu_c_s2;
   endcase
end



assign rf_c = dest_bus_s2;

// Write back - destination selection
// see wb_dest.generated.v
always @* begin
   rf_write = 1'b0;
   write_ar = 3'b0;
       mem8_write = 1'b0;
       mem16_write = 1'b0;
       rf_write = 1'b0;
       mem_we <= 1'b0;


       case(dec_writeback_dest_mux_sel)
     USEQ_SEL_DEST_RFC: begin
                rf_write =  1'b1;
     end
     USEQ_SEL_DEST_COND_LINK_REG: begin
     end
     USEQ_SEL_DEST_MEM16: begin
       // Fixme
     end
     USEQ_SEL_DEST_MEM8: begin
       // Fixme
     end
     USEQ_SEL_DEST_SR: begin
       case(dec_rd)
        USEQ_SR_AR0 : begin
        end
        USEQ_SR_AR1 : begin
        end
        USEQ_SR_AR2 : begin
```

```
            end
             default: begin
             end

          endcase
        end
        default: begin
                    rf_write = 1'b0;
        end
      endcase

case(dec_writeback_dest_mux_sel_s1)
        USEQ_SEL_DEST_RFC: begin
        end
        USEQ_SEL_DEST_COND_LINK_REG: begin
        end
        USEQ_SEL_DEST_MEM16: begin
                    mem16_write = 1'b1;
                    mem_we <= 1'b1;
        end
        USEQ_SEL_DEST_MEM8: begin
                    mem8_write = 1'b1;
          mem_we <= 1'b1;
        end
        USEQ_SEL_DEST_SR: begin
          case(dec_rd)
           USEQ_SR_AR0 : begin
           end
           USEQ_SR_AR1 : begin
           end
           USEQ_SR_AR2 : begin
           end
            default: begin

            end

          endcase
        end
        default: begin
                      mem8_write = 1'b0;
            mem8_write = 1'b0;
```

```
            mem_we <= 1'b0;
        end
      endcase

case(dec_writeback_dest_mux_sel_s2)
        USEQ_SEL_DEST_RFC: begin
        end
        USEQ_SEL_DEST_COND_LINK_REG: begin
        end
        USEQ_SEL_DEST_MEM16: begin
          // Fixme

        end
        USEQ_SEL_DEST_MEM8: begin

          // Fixme
        end
        USEQ_SEL_DEST_SR: begin
          case(dec_rd)
           USEQ_SR_AR0 : begin
             write_ar = 3'b001;
           end
           USEQ_SR_AR1 : begin
             write_ar = 3'b010;
           end
           USEQ_SR_AR2 : begin
             write_ar = 3'b100;
           end
            default: begin
              write_ar = 3'b000;
            end

          endcase
        end
        default: begin

          write_ar = 3'b0;
        end
      endcase
  end
```

```
// Selection of special register for read
// To be used by mfrs instruction

always @* begin
  case(dec_rs1)
    USEQ_SR_AR0 : begin
      sr_read_bus = ar0_r;
    end
    USEQ_SR_AR1 : begin
      sr_read_bus = ar1_r;
    end
    USEQ_SR_AR2 : begin
      sr_read_bus = ar2_r;
    end
    USEQ_SR_SR : begin
      sr_read_bus = 0; // Fixme
    end
    default: begin
    end

  endcase
end

// Halt instruction
always @* begin
  case(dec_flow_halt_mux_sel)
    USEQ_FLOW_HALT_YES: begin
      halt = 1'b1;
    end
    USEQ_FLOW_HALT_NO: begin
      halt = 1'b0;
    end
    default: begin
      halt = 1'b0;
    end
  endcase
end
```

```
assign decode_string = {dec_opcode1,dec_opcode3,
   dec_opcode4,dec_cond};
reg [39:0]          _codeop_ascii_r;          //
   Decode of state_r
initial
  _codeop_ascii_r = "Unknown";


always @* begin
  // FIXME : we need the default case
  casez(decode_string)
  //@begin[case_inst]
  USEQ_OPCODES_AND: begin
       _codeop_ascii_r = "USEQ_OPCODES_AND";
       dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
       dec_writeback_source_mux_sel =
          USEQ_SEL_WBSOURCE_ALU;
       dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
       dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
       dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
       dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
       dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
       dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
       dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
       dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
  end
  USEQ_OPCODES_XNORR1: begin
       _codeop_ascii_r = "USEQ_OPCODES_XNORR1";
       dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
       dec_writeback_source_mux_sel =
          USEQ_SEL_WBSOURCE_ALU;
       dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
       dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
       dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
       dec_alu_sourceb_mux_sel =
          USEQ_SEL_ALUB_RFB_OR_UK4;
       dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
       dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
       dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
       dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
  end
```

```
USEQ_OPCODES_ADD2C: begin
      _codeop_ascii_r = "USEQ_OPCODES_ADD2C";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_LDIR1: begin
      _codeop_ascii_r = "USEQ_OPCODES_LDIR1";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_R1;
      dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K12;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
      dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_ORR1: begin
      _codeop_ascii_r = "USEQ_OPCODES_ORR1";
      dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
      dec_writeback_source_mux_sel =
         USEQ_SEL_WBSOURCE_ALU;
      dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
      dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
      dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
      dec_alu_sourceb_mux_sel =
         USEQ_SEL_ALUB_RFB_OR_UK4;
      dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
      dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
      dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
```

```
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_HALT: begin
        _codeop_ascii_r = "USEQ_OPCODES_HALT";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_YES;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XORR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_XORR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_LB: begin
        _codeop_ascii_r = "USEQ_OPCODES_LB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM8;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
```

```
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB2C: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2C";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_NOT_RFB
            ;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_RET: begin
        _codeop_ascii_r = "USEQ_OPCODES_RET";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_PC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R14;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_FROM_ALU;
    end
    USEQ_OPCODES_LW: begin
        _codeop_ascii_r = "USEQ_OPCODES_LW";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_MEM16;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
```

```
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SUB3: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SUB;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_XNOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XNOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XNOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_BRPC: begin
        _codeop_ascii_r = "USEQ_OPCODES_BRPC";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_COND_LINK_REG;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
```

```
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RS2;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K8;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_NPC;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_SUB2: begin
        _codeop_ascii_r = "USEQ_OPCODES_SUB2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_NOT_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ANDR1: begin
        _codeop_ascii_r = "USEQ_OPCODES_ANDR1";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_AND;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MFSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MFSR";
```

```
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_SR;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_MTSR: begin
        _codeop_ascii_r = "USEQ_OPCODES_MTSR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_SR;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ORI4: begin
        _codeop_ascii_r = "USEQ_OPCODES_ORI4";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
           USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
           USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
```

```
USEQ_OPCODES_SRAI: begin
    _codeop_ascii_r = "USEQ_OPCODES_SRAI";
    dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_ALU;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
    dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
    dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
    dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_SEXT: begin
    _codeop_ascii_r = "USEQ_OPCODES_SEXT";
    dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_ALU;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
    dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SEXT;
    dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
    dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
end
USEQ_OPCODES_BR: begin
    _codeop_ascii_r = "USEQ_OPCODES_BR";
    dec_writeback_dest_mux_sel =
      USEQ_SEL_DEST_COND_LINK_REG;
    dec_writeback_source_mux_sel =
      USEQ_SEL_WBSOURCE_ALU;
    dec_rf_sela_mux_sel = USEQ_SEL_RFA_R0;
    dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
    dec_rf_selc_mux_sel = USEQ_SEL_RFC_R0;
    dec_alu_sourceb_mux_sel =
      USEQ_SEL_ALUB_RFB_OR_UK4;
    dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
    dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
```

```
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel =
            USEQ_SEL_PC_COND_LOAD_FROM_ALU;
    end
    USEQ_OPCODES_XOR: begin
        _codeop_ascii_r = "USEQ_OPCODES_XOR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_XOR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRA: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRA";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRA;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD3: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD3";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
```

```
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_ADD2: begin
        _codeop_ascii_r = "USEQ_OPCODES_ADD2";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel =
            USEQ_SEL_ALUB_RFB_OR_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SW: begin
        _codeop_ascii_r = "USEQ_OPCODES_SW";
        dec_writeback_dest_mux_sel =
            USEQ_SEL_DEST_MEM16;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
```

```
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SRL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SRL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SRL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLL: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLL";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SLLI: begin
        _codeop_ascii_r = "USEQ_OPCODES_SLLI";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
```

```
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_R1;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_UK4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_SLL;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_SB: begin
        _codeop_ascii_r = "USEQ_OPCODES_SB";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_MEM8
            ;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_RFB;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RS1;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RD;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_K4;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_ADD;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
    USEQ_OPCODES_OR: begin
        _codeop_ascii_r = "USEQ_OPCODES_OR";
        dec_writeback_dest_mux_sel = USEQ_SEL_DEST_RFC;
        dec_writeback_source_mux_sel =
            USEQ_SEL_WBSOURCE_ALU;
        dec_rf_sela_mux_sel = USEQ_SEL_RFA_RD;
        dec_rf_selb_mux_sel = USEQ_SEL_RFB_RS2;
        dec_rf_selc_mux_sel = USEQ_SEL_RFC_RD;
        dec_alu_sourceb_mux_sel = USEQ_SEL_ALUB_RFB;
        dec_alu_sourcea_mux_sel = USEQ_SEL_ALUA_RFA;
        dec_alu_op_mux_sel = USEQ_SEL_ALUOP_OR;
        dec_flow_halt_mux_sel = USEQ_FLOW_HALT_NO;
        dec_pc_next_mux_sel = USEQ_SEL_PC_INCREMENT;
    end
   //@end[case_inst]
   endcase
end
```

```
  wire cond_branch;
  assign cond_branch = 1'b1; // Fixme

always @(posedge pc_r_clk or negedge rst_n) begin
   if(rst_n == 1'b0) begin
     /*AUTORESET*/
     // Beginning of autoreset for uninitialized flops
         stall_cnt <= {(3){1'b0}};
         npc <= 0;
     pc_r <= {(1+(USEQ_WORD_MSB)){1'b0}};
         stalling <= 0;
     // End of automatics
         //rf_req<=0 ;
   end
   else begin
               if(halt == 1'b1) begin
                       pc_r <= pc_r;
               end else begin
                       if(stall == 1'b1) begin
                               /*if(stall_cnt<
                                  stall_target) begin
                                       stalling <= 1;
                                       stall_cnt <=
                                          stall_cnt+1;
                                       pc_r <= pc_r;
                               end else begin
                                       stalling <= 0;
                                       stall_cnt <= 0;
                                       pc_r <= pc_r +
                                          USEQ_PC_INC;
                               end*/
                               pc_r <= pc_r;
                       end else begin

                                case(
                                   dec_pc_next_mux_sel
                                   )
                                 USEQ_SEL_PC_INCREMENT
                                    : begin
```

```
                    pc_r <= pc_r +

                        USEQ_PC_INC
                        ;
                end
                USEQ_SEL_PC_FROM_ALU:
                    begin
                        // pc_r <=
                            dest_bus;
                end
                USEQ_SEL_PC_COND_LOAD_FROM_ALU
                    : begin
                            if(
                                cond_branch
                                )
                                begin

                        //    pc_r <=
                            dest_bus;
                                end
                                else
                                    begin

                                    pc_r
                                        <=

                                        pc_r
                                        +

                                        USEQ_PC_INC
                                        ;
                                end
                        end

                default: begin
                        pc_r <= pc_r +
                            USEQ_PC_INC;
                end

            endcase
        end
```

```
        //pc_r <= npc;
        halt_r <= halt;

                end
          end // always @ (posedge clk or  rst_n)
end


  assign carry_in = 0; // FIXME

  pulse_generator PGRAM (ram_clk,ram_ack);
  always @* begin
        if(rst_n == 1'b0) begin
                ram_req_pre =0;
        end
        else begin
                ram_req_pre = !alu_ack;
        end
  end

  always @* begin
   if(rst_n == 1'b0) begin
                rf_req = 0;
        end else begin
                casez(decode_string)
                USEQ_OPCODES_SW: begin
                        rf_req = mem_req_delayed;
                end
                USEQ_OPCODES_SB: begin
                        rf_req = mem_req_delayed;
                end
                USEQ_OPCODES_LB:  begin
                        rf_req = mem_req_delayed;
                end
                USEQ_OPCODES_LW: begin
                        rf_req = mem_req_delayed;
                end
                default: begin
                 rf_req = rf_req_pre;
                end
                endcase
```

```
                end
 end


always @* begin
        if(rst_n == 1'b0) begin
                rf_req_pre = 0;
                mem_req = 0;
                mem_clk = 0;
        end
        else begin
                casez(decode_string)
                USEQ_OPCODES_SW: begin
                        mem_req = mem_req_pre;
                        if(stall == 0) begin
                                mem_clk = mem_clk_pre;
                        end else begin
                                        mem_clk = 1'b0;
                                end
                end
                USEQ_OPCODES_SB: begin
                        mem_req = mem_req_pre;
                        if(stall == 0) begin
                                mem_clk = mem_clk_pre;
                        end else begin
                                        mem_clk = 1'b0;
                                end
                end
                USEQ_OPCODES_LB:  begin
                        mem_req = mem_req_pre;
                        if(stall == 0) begin
                                mem_clk = mem_clk_pre;
                        end else begin
                                        mem_clk = 1'b0;
                                end
                end
                USEQ_OPCODES_LW: begin
                        mem_req = mem_req_pre;
                        if(stall == 0) begin
                                mem_clk = mem_clk_pre;
                        end else begin
```

```
                                                mem_clk = 1'b0;
                                        end
                    end
                    default: begin
                     mem_req = 0;
                     rf_req_pre = mem_req_pre;
                     mem_clk = 0;
                    end
                    endcase
            end
end


  simd_useq_alu U_ALU (
                        // Outputs
                        .c                  (alu_c[
                           USEQ_WORD_MSB:0]),
                        // Inputs
                        .a                  (alu_a_s0[
                           USEQ_WORD_MSB:0]),
                        .b                  (alu_b_s0[
                           USEQ_WORD_MSB:0]),
                        .op                 (
                           dec_alu_op_mux_sel_s0[
                           USEQ_SEL_ALUOP_MSB:0]),
                        .carry_in           (carry_in),
                        .carry_out          (carry_out));

   /* simd_useq_rf AUTO_TEMPLATE(
    ); */
  simd_useq_rf U_RF (
                      .rf_a                 (rf_a_out[
                         USEQ_WORD_MSB:0]),
                      .rf_b                 (rf_b_out[
                         USEQ_WORD_MSB:0]),
                      /*AUTOINST*/
                      // Inputs
                      .rf_sela              (rf_sela[
                         USEQ_RF_SEL_MSB:0]),
                      .rf_selb              (rf_selb[
                         USEQ_RF_SEL_MSB:0]),
```

```
                .rf_selc            (rf_selc_s2[
                  USEQ_RF_SEL_MSB:0]),
                .rf_c               (rf_c[
                  USEQ_WORD_MSB:0]),
                .rf_write           (rf_write_s2),
                .clk                (rf_clk),
                .rst_n              (rst_n));



endmodule // simd_useq
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

K.Q - Asynchronous stage CPU FPGA modified files for synthesis

Register file

```
module simd_useq_rf (/*AUTOARG*/
  // Outputs
  rf_a, rf_b, rf_reg,
  // Inputs
  rf_sela, rf_selb, rf_selc, rf_c, rf_write, clk, rst_n
    , rf_selreg
  );
`include "simd_params.v"
  parameter NUM_REGS=15;

  // Selection
  input  [USEQ_RF_SEL_MSB:0] rf_sela;
  input [USEQ_RF_SEL_MSB:0]  rf_selb;
  input [USEQ_RF_SEL_MSB:0]   rf_selc;
  input [USEQ_RF_SEL_MSB:0]   rf_selreg;
  // Data ports
  output [USEQ_WORD_MSB:0] rf_a;
```

```
  output [USEQ_WORD_MSB:0] rf_b;
  output reg [USEQ_WORD_MSB:0] rf_reg;
  input [USEQ_WORD_MSB:0]  rf_c;

  input                    rf_write;
  input                    clk;
  input                    rst_n;

  /*AUTOINPUT*/

  /*AUTOOUTPUT*/

  /*AUTOREG*/
  // Beginning of automatic regs (for this module's
     undeclared outputs)
  reg [USEQ_WORD_MSB:0] rf_a;
  reg [USEQ_WORD_MSB:0] rf_b;
  // End of automatics

  /*AUTOWIRE*/
wire  [31:0] r0 ;
wire  [31:0] r1 ;
wire  [31:0] r2 ;
wire  [31:0] r3 ;
wire  [31:0] r4 ;
wire  [31:0] r5 ;
wire  [31:0] r6 ;
wire  [31:0] r7 ;
wire  [31:0] r8 ;
wire  [31:0] r9 ;
wire  [31:0] r10;
wire  [31:0] r11;
wire  [31:0] r12;
wire  [31:0] r13;
wire  [31:0] r14;



  reg [USEQ_WORD_MSB:0] regfile [1:NUM_REGS-1];
        //Loooong sensitivity list due to bug in xise
```

```
        always @(regfile[1],regfile[2],regfile[3],
           regfile[4],regfile[5],regfile[6],regfile[7],
           regfile[8],regfile[9],regfile[10],regfile
           [11],regfile[12],regfile[13],regfile[14])
           begin
                  if(rf_sela != 0) begin
                         rf_a = regfile[rf_sela];
                  end
                  else begin
                         rf_a = 0;
                  end
          end
        always @(regfile[1],regfile[2],regfile[3],
           regfile[4],regfile[5],regfile[6],regfile[7],
           regfile[8],regfile[9],regfile[10],regfile
           [11],regfile[12],regfile[13],regfile[14])
           begin
                  if(rf_selb != 0) begin
                         rf_b = regfile[rf_selb];
                  end
                  else begin
                         rf_b = 0;
                  end
          end

        always @(regfile[1],regfile[2],regfile[3],
           regfile[4],regfile[5],regfile[6],regfile[7],
           regfile[8],regfile[9],regfile[10],regfile
           [11],regfile[12],regfile[13],regfile[14])
           begin
                  if(rf_selreg != 0) begin
                         rf_reg = regfile[rf_selreg];
                  end
                  else begin
                         rf_reg = 0;
                  end
          end
  integer k;

  always @(posedge clk or negedge rst_n) begin
        if(rst_n == 0) begin
```

```
                /*for (k = 1; k < NUM_REGS; k = k + 1)
                    begin
                        regfile[k] <= 0;
                    end*/
    end else if((rf_selc != 0) && rf_write) begin
      regfile[rf_selc] <= rf_c;
    end
  end


  // For debugging

// synthesis translate_off




assign      r1 =        regfile[1];
assign      r2 =        regfile[2];
assign      r3 =        regfile[3];
assign      r4 =        regfile[4];
assign      r5 =        regfile[5];
assign      r6 =        regfile[6];
assign      r7 =        regfile[7];
assign      r8 =        regfile[8];
assign      r9 =        regfile[9];
assign      r10 =       regfile[10];
assign      r11 =       regfile[11];
assign      r12 =       regfile[12];
assign      r13 =       regfile[13];
assign      r14 =       regfile[14];



// synthesis translate_on
```

```
endmodule // dlx3_regfile
/*
 Local Variables:
 verilog-library-directories:(
 "."
 )
 End:
 */
```

Delay modules dmfpga.v

```
`timescale 1ns / 1ps
module delay_module(out,in,sel) ;
        function integer clog2;
                input integer value;
                begin
                        value = value-1;
                        for (clog2=0; value>0; clog2=
                           clog2+1)
                                value = value>>1;
                end
        endfunction


        parameter DELAY = 4;
        parameter NR_BITS = clog2(DELAY) -1 ;
        input in;
        input [NR_BITS:0] sel;
        output out;
        reg out;
        //ORDLY15s8fhvt DELAYEL(out,in,1'b1);
        wire [DELAY-1:0] in_temp;
        wire test;
        wire zero;
        assign zero = 1'b0;
        genvar i;
                ordelayelement DELAYEL(.out(in_temp[0])
                   ,.a(in),.b(zero));
        generate
```

```
                for(i=0;i<DELAY-1; i=i+1) begin:
                    delaygen
                                ordelayelement DELAYEL
                                    (.out(in_temp[i+1])
                                    ,.a(in_temp[i]) ,.b(
                                    zero));
                end
        endgenerate

        always @ * begin
                out     <= in_temp[sel];
        end


endmodule
```

K.R - Asynchronous CPU Testbench

```
`timescale 1ns/1ps


module tb_simd (/*AUTOARG*/);

`include "simd_params.v"



  /*AUTOINPUT*/

  /*AUTOOUTPUT*/

  /*AUTOREG*/

  /*AUTOWIRE*/
  // Beginning of automatic wires (for undeclared
     instantiated-module outputs)
```

```
  wire                    clk;                    // From
      U_CLOK_GEN of clock_gen.v
  wire [USEQ_WORD_MSB:0] pc_r;                    // From
      U_SIMD of simd.v
  wire                    rst_n;                  // From
      U_RESET_GEN of reset_gen.v
  // End of automatics
  wire ram_clk;
  wire cpu_ack;

parameter RAM_SIZE = 1536;
  reg [USEQ_INST_MSB:0] instruction_reg;
  wire [USEQ_INST_MSB:0] instruction;


        wire me;
        wire re;
        reg [3:0] we;
        reg [3:0]rm;
        reg [15:0]instruction_in;
        reg pd;
        reg pdd;
        reg [15:0]      mem0 [0:RAM_SIZE-1];
        reg load_program;
        reg started_prog;
        reg [10:0]count;
        reg [3:0] sel_mode;
   /* simd AUTO_TEMPLATE(
    .instruction              (instruction_reg[
       USEQ_INST_MSB:0]),
    ); */
  simd U_SIMD (
                          /*AUTOINST*/
               // Outputs
               .pc_r                    (pc_r[
                  USEQ_WORD_MSB:0]),
                                .cpu_ack
                                        (cpu_ack),
                                .ram_clk
                                        (ram_clk),
               // Inputs
```

```
                                .sel_mode
                                                (
                                   sel_mode[3:0]),
                                         .rst_n

                                                (rst_n),
                        .instruction            (
                           instruction_reg[USEQ_INST_MSB:0]), //
                            Templated
                                .cpu_req
                                          (cpu_req),
                                .clk
                                          (clk)
                                );


  /* ram AUTO_TEMPLATE(
   .addr                  (pc_r[USEQ_WORD_MSB:1]), //
      Fixme - truncation...
   .we                    (1'b0),
   .write_data            ({(USEQ_WORD_BITS){1'b0}}),
    .read_data             (instruction[USEQ_INST_MSB
       :0]),
   .clock                 (clk),
   .cs                    (1'b1),
       ); */
/*ram #(.WORD_WIDTH(USEQ_WORD_BITS),
       .ADDR_WIDTH(USEQ_WORD_BITS-1))
U_USEQ_CODE_RAM (
                 .read_data             (instruction[
                    USEQ_INST_MSB:0]),
                 // Inputs
                 .clock                 (clk),
                                        // Templated
                 .addr                  (pc_r[
                    USEQ_WORD_MSB:1]), // Templated
                 .write_data            ({(
                    USEQ_WORD_BITS){1'b0}}), //
                    Templated
                 .cs                    (1'b1),
                                       // Templated
```

```
                .we                     (1'b0));
                                       // Templated


*/ prog_mem_wrapper PROGMEMWRAP (
        .load_program (load_program),
        .prog_clk (prog_clk),
        .prog_adr (count[10:0]),
        .clk     (clk),
        .pc_r    (pc_r[15:0]),
        .instruction_in (instruction_in[15:0]),
        .instruction     (instruction[USEQ_INST_MSB:0]),
        .rst_n_in (rst_n_temp),
        .rst_n_out (rst_n));

always @* begin
  instruction_reg <= instruction;
end
initial begin
  instruction_reg <= 0;

end




//    /* ram16_8 AUTO_TEMPLATE(
//   ); */
//ram16_8 U_USEQ_DATA_RAM (
//                         /*AUTOINST*/
//                          // Outputs
//                          .read_data            (
   read_data[15:0]),
//                          // Inputs
//                          .clock                (clock)
   ,
//                          .addr                 (addr[
   ADDR_MSB:0]),
//                          .write_data           (
   write_data[15:0]),
```

```
//                              .cs                     (cs),
//                              .we                     (we
   [1:0]));
//
//
//
```

```
    /* clock_gen AUTO_TEMPLATE(
    ); */
  clock_gen U_CLOK_GEN (
                           /*AUTOINST*/
                          // Outputs
                          .clk              (clk));

 clock_gen PROG_CLOK_GEN (
                           /*AUTOINST*/
                          // Outputs
                          .clk              (prog_clk));
```

```
   /* reset_gen AUTO_TEMPLATE(
    .reset_n                (rst_n),
    ); */
  reset_gen U_RESET_GEN (
                           /*AUTOINST*/
                          // Outputs
                          .reset_n                (
                             rst_n_temp));          //
                             Templated
```

```
  task load_program_memory;
    reg [1024:0] filename;
    reg [7:0]    memory [2<<USEQ_ADDR_BITS:0]; // 8-bit
        memory
```

```
    integer         i;
    integer         dummy;
    begin

      filename = 0;
      dummy = $value$plusargs("program_memory=%s",
          filename);
      if(filename ==0) begin
        $display("WARNING! No content specified for
            program memory");
        end
      else begin
        $readmemh (filename, memory);
        for(i=0; i<(1<<USEQ_ADDR_BITS); i=i+2) begin
         mem0[i/2] = {memory[i+1],memory[i]};
        end
      end
    end
  endtask // load_program_memory

  event evt_finish;

  initial
    begin
      load_program_memory;
          `ifdef NTL
      $dumpfile("tb_simdNTL.vcd");
      $dumpvars(0,tb_simd);
          `else
          $dumpfile("tb_simd.vcd");
      $dumpvars(0,tb_simd);
          `endif

    end
        always @(posedge prog_clk or negedge rst_n_temp
            ) begin
        if(rst_n_temp == 1'b0) begin
                  started_prog <= 0;
                  count <=0;
                  instruction_in <= 0;
        end else
```

```
        if((load_program == 1'b1) && (started_prog ==1'
           b0)) begin
                count <= 0;
                started_prog <=1;
        end else if ((load_program ==1'b1) && (
           started_prog ==1'b1) && (count<30)) begin
                instruction_in <= mem0[count][15:0];
                count <= count + 1;
        end else begin
                load_program<=0;
                count <= 0;
                started_prog <=0;
                instruction_in <= 0;
        end
        end

  initial begin
        sel_mode = 4'b0;
        load_program = 1;
    #50000;
    $display("-E- Timeout reached !");
    -> evt_finish;
    #10;
    $finish;
  end

`ifdef NTL
  reg halt;
  always @* begin
  if (instruction == 16'hE003) begin
        halt = 1;
        #100;
        end else if(halt==1) begin
    $display("-I- Simulation halted by the CPU - <halt>
       instruction");
    -> evt_finish;
    #10;
    $finish;
        end else begin
                halt = 0;
        end
```

```
        end


    integer i;
    always @(evt_finish) begin
        $display("r00=0x%04h r01=0x%04h r02=0x%04h r03=0
            x%04h ",
                 0,
                 U_SIMD.U_USEQ.U_RF.regfile[223:208],
                 U_SIMD.U_USEQ.U_RF.regfile[207:192],
                 U_SIMD.U_USEQ.U_RF.regfile[191:176]);
        $display("r04=0x%04h r05=0x%04h r06=0x%04h r07=0
            x%04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[175:160],
                 U_SIMD.U_USEQ.U_RF.regfile[159:144],
                 U_SIMD.U_USEQ.U_RF.regfile[143:128],
                 U_SIMD.U_USEQ.U_RF.regfile[127:112]);
        $display("r08=0x%04h r09=0x%04h r10=0x%04h r11=0
            x%04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[111:96],
                 U_SIMD.U_USEQ.U_RF.regfile[95:80],
                 U_SIMD.U_USEQ.U_RF.regfile[79:64],
                 U_SIMD.U_USEQ.U_RF.regfile[63:48]);
        $display("r12=0x%04h r13=0x%04h r14=0x%04h pc=0x
            %04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[47:32],
                 U_SIMD.U_USEQ.U_RF.regfile[31:16],
                 U_SIMD.U_USEQ.U_RF.regfile[15:0],
                 U_SIMD.U_USEQ.pc_r);
      end
`else
  always @(posedge U_SIMD.U_USEQ.halt) begin
    $display("-I- Simulation halted by the CPU - <halt>
        instruction");
    -> evt_finish;
    #10;
    $finish;
  end


    integer i;
```

```
    always @(evt_finish) begin
        $display("r00=0x%04h r01=0x%04h r02=0x%04h r03=0
            x%04h ",
                0,
                U_SIMD.U_USEQ.U_RF.regfile[1],
                U_SIMD.U_USEQ.U_RF.regfile[2],
                U_SIMD.U_USEQ.U_RF.regfile[3]);
        $display("r04=0x%04h r05=0x%04h r06=0x%04h r07=0
            x%04h ",
                U_SIMD.U_USEQ.U_RF.regfile[4],
                U_SIMD.U_USEQ.U_RF.regfile[5],
                U_SIMD.U_USEQ.U_RF.regfile[6],
                U_SIMD.U_USEQ.U_RF.regfile[7]);
        $display("r08=0x%04h r09=0x%04h r10=0x%04h r11=0
            x%04h ",
                U_SIMD.U_USEQ.U_RF.regfile[8],
                U_SIMD.U_USEQ.U_RF.regfile[9],
                U_SIMD.U_USEQ.U_RF.regfile[10],
                U_SIMD.U_USEQ.U_RF.regfile[11]);
        $display("r12=0x%04h r13=0x%04h r14=0x%04h pc=0x
            %04h ",
                U_SIMD.U_USEQ.U_RF.regfile[12],
                U_SIMD.U_USEQ.U_RF.regfile[13],
                U_SIMD.U_USEQ.U_RF.regfile[14],
                U_SIMD.U_USEQ.pc_r);
    end
`endif


endmodule // tb_pipeline
/*
 Local Variables:
 verilog-library-directories:(
 "."
 "../../rtl/module/verilog"
 )
 End:
 */
```

K.S - Synchronous CPU Testbench

```
`timescale 1ns/1ps


module tb_simd (/*AUTOARG*/);

`include "simd_params.v"



  /*AUTOINPUT*/

  /*AUTOOUTPUT*/

  /*AUTOREG*/

  wire                    clk;                    // From
      U_CLOK_GEN of clock_gen.v
  wire [USEQ_WORD_MSB:0] pc_r;                    // From
      U_SIMD of simd.v
  wire                    rst_n;                  // From
      U_RESET_GEN of reset_gen.v
  wire                    rst_n_temp;
  // End of automatics

parameter RAM_SIZE = 1536;
  reg [USEQ_INST_MSB:0] instruction_reg;
  wire [USEQ_INST_MSB:0] instruction;


        wire me;
        wire re;
        reg [3:0] we;
        reg [3:0]rm;
        reg [15:0]instruction_in;
        reg pd;
        reg pdd;
        reg [15:0]      mem0 [0:RAM_SIZE-1];
        reg load_program;
```

```
      reg started_prog;
      reg [10:0]count;
      reg [3:0] sel_mode;
 /* simd AUTO_TEMPLATE(
  .instruction              (instruction_reg[
     USEQ_INST_MSB:0]),
  ); */


simd U_SIMD (
                          /*AUTOINST*/
             // Outputs
             .pc_r                      (pc_r[
                USEQ_WORD_MSB:0]),
             // Inputs
             .clk                       (clk),
             .instruction               (
                instruction_reg[USEQ_INST_MSB:0]), //
                 Templated
             .rst_n                     (rst_n)
                        .sel_mode
                                        (sel_mode
                           [3:0]));


 /* ram AUTO_TEMPLATE(
  .addr                     (pc_r[USEQ_WORD_MSB:1]), //
     Fixme - truncation...
  .we                       (1'b0),
  .write_data               ({(USEQ_WORD_BITS){1'b0}}),
   .read_data                (instruction[USEQ_INST_MSB
     :0]),
  .clock                    (clk),
  .cs                       (1'b1),
      ); */
/*ram #(.WORD_WIDTH(USEQ_WORD_BITS),
      .ADDR_WIDTH(USEQ_WORD_BITS-1))
U_USEQ_CODE_RAM (
                 .read_data             (instruction[
                    USEQ_INST_MSB:0]),
                 // Inputs
```

```
                    .clock                  (clk),
                                            // Templated
                    .addr                   (pc_r[
                      USEQ_WORD_MSB:1]), // Templated
                    .write_data             ({(
                      USEQ_WORD_BITS){1'b0}}), //
                      Templated
                    .cs                     (1'b1),
                                           // Templated
                    .we                     (1'b0));
                                           // Templated


  */ prog_mem_wrapper PROGMEMWRAP (
        .load_program (load_program),
        .prog_clk (prog_clk),
        .prog_adr (count[10:0]),
        .clk     (clk),
        .pc_r    (pc_r[15:0]),
        .instruction_in (instruction_in[15:0]),
        .instruction     (instruction[USEQ_INST_MSB:0]),
        .rst_n_in (rst_n_temp),
        .rst_n_out (rst_n));

  always @* begin
    instruction_reg <= instruction;
  end
  initial begin
    instruction_reg <= 0;

  end




//    /* ram16_8 AUTO_TEMPLATE(
//   ); */
//ram16_8 U_USEQ_DATA_RAM (
//                          /*AUTOINST*/
//                           // Outputs
```

```
//                              .read_data              (
   read_data[15:0]),
//                              // Inputs
//                              .clock                  (clock)
   ,
//                              .addr                   (addr[
   ADDR_MSB:0]),
//                              .write_data             (
   write_data[15:0]),
//                              .cs                     (cs),
//                              .we                     (we
   [1:0]));
//
//
//
```

```
    /* clock_gen AUTO_TEMPLATE(
    ); */
  clock_gen U_CLOK_GEN (
                            /*AUTOINST*/
                           // Outputs
                           .clk             (clk));

clock_gen PROG_CLOK_GEN (
                            /*AUTOINST*/
                           // Outputs
                           .clk             (prog_clk));
```

```
   /* reset_gen AUTO_TEMPLATE(
    .reset_n                (rst_n),
    ); */
  reset_gen U_RESET_GEN (
                            /*AUTOINST*/
                           // Outputs
                           .reset_n                 (
                              rst_n_temp));          //
```

Templated

```
task load_program_memory;
  reg [1024:0] filename;
  reg [7:0]    memory [2<<USEQ_ADDR_BITS:0]; // 8-bit
      memory
  integer      i;
  integer      dummy;
  begin

    filename = 0;
    dummy = $value$plusargs("program_memory=%s",
        filename);
    if(filename ==0) begin
      $display("WARNING! No content specified for
          program memory");
      end
    else begin
      $readmemh (filename, memory);
      for(i=0; i<(1<<USEQ_ADDR_BITS); i=i+2) begin
       mem0[i/2] = {memory[i+1],memory[i]};
      end
    end
  end
endtask // load_program_memory

event evt_finish;

initial
  begin
    load_program_memory;
        `ifdef NTL
    $dumpfile("tb_simdNTL.vcd");
    $dumpvars(0,tb_simd);
        `else
        $dumpfile("tb_simd.vcd");
    $dumpvars(0,tb_simd);
        `endif
```

```
    end
        always @(posedge prog_clk or negedge rst_n_temp
            ) begin
        if(rst_n_temp == 1'b0) begin
                    started_prog <= 0;
                    count <=0;
                    instruction_in <= 0;
        end else
        if((load_program == 1'b1) && (started_prog ==1'
            b0)) begin
                    count <= 0;
                    started_prog <=1;
        end else if ((load_program ==1'b1) && (
            started_prog ==1'b1) && (count<30)) begin
                    instruction_in <= mem0[count][15:0];
                    count <= count + 1;
        end else begin
                    load_program<=0;
                    count <= 0;
                    started_prog <=0;
                    instruction_in <= 0;
        end
        end

  initial begin
        load_program = 1;
        sel_mode = 4'b0;
    #5000;
    $display("-E- Timeout reached !");
    -> evt_finish;
    #10;
    $finish;
  end

`ifdef NTL
  reg halt;
  always @(negedge clk) begin
  if (instruction == 16'hE003) begin
        halt = 1;
        end else if(halt==1) begin
```

```
    $display("-I- Simulation halted by the CPU - <halt>
       instruction");
    -> evt_finish;
    #10;
    $finish;
        end else begin
                halt = 0;
        end
        end


    integer i;
    always @(evt_finish) begin
        $display("r00=0x%04h r01=0x%04h r02=0x%04h r03=0
           x%04h ",
                0,
                U_SIMD.U_USEQ.U_RF.regfile[223:208],
                U_SIMD.U_USEQ.U_RF.regfile[207:192],
                U_SIMD.U_USEQ.U_RF.regfile[191:176]);
        $display("r04=0x%04h r05=0x%04h r06=0x%04h r07=0
           x%04h ",
                U_SIMD.U_USEQ.U_RF.regfile[175:160],
                U_SIMD.U_USEQ.U_RF.regfile[159:144],
                U_SIMD.U_USEQ.U_RF.regfile[143:128],
                U_SIMD.U_USEQ.U_RF.regfile[127:112]);
        $display("r08=0x%04h r09=0x%04h r10=0x%04h r11=0
           x%04h ",
                U_SIMD.U_USEQ.U_RF.regfile[111:96],
                U_SIMD.U_USEQ.U_RF.regfile[95:80],
                U_SIMD.U_USEQ.U_RF.regfile[79:64],
                U_SIMD.U_USEQ.U_RF.regfile[63:48]);
        $display("r12=0x%04h r13=0x%04h r14=0x%04h pc=0x
           %04h ",
                U_SIMD.U_USEQ.U_RF.regfile[47:32],
                U_SIMD.U_USEQ.U_RF.regfile[31:16],
                U_SIMD.U_USEQ.U_RF.regfile[15:0],
                U_SIMD.U_USEQ.pc_r);
      end

`else
always @(posedge U_SIMD.U_USEQ.halt_r) begin
```

```
      $display("-I- Simulation halted by the CPU - <halt>
         instruction");
      -> evt_finish;
      #10;
      $finish;
    end


    integer i;
    always @(evt_finish) begin
        $display("r00=0x%04h r01=0x%04h r02=0x%04h r03=0
           x%04h ",
                 0,
                 U_SIMD.U_USEQ.U_RF.regfile[1],
                 U_SIMD.U_USEQ.U_RF.regfile[2],
                 U_SIMD.U_USEQ.U_RF.regfile[3]);
        $display("r04=0x%04h r05=0x%04h r06=0x%04h r07=0
           x%04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[4],
                 U_SIMD.U_USEQ.U_RF.regfile[5],
                 U_SIMD.U_USEQ.U_RF.regfile[6],
                 U_SIMD.U_USEQ.U_RF.regfile[7]);
        $display("r08=0x%04h r09=0x%04h r10=0x%04h r11=0
           x%04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[8],
                 U_SIMD.U_USEQ.U_RF.regfile[9],
                 U_SIMD.U_USEQ.U_RF.regfile[10],
                 U_SIMD.U_USEQ.U_RF.regfile[11]);
        $display("r12=0x%04h r13=0x%04h r14=0x%04h pc=0x
           %04h ",
                 U_SIMD.U_USEQ.U_RF.regfile[12],
                 U_SIMD.U_USEQ.U_RF.regfile[13],
                 U_SIMD.U_USEQ.U_RF.regfile[14],
                 U_SIMD.U_USEQ.pc_r);
      end
        `endif

endmodule // tb_pipeline
/*
 Local Variables:
 verilog-library-directories:(
```

```
 "."
 "../../rtl/module/verilog"
 )
 End:
 */
```

K.T - Test program examples ASM

```
# Desc: Test zjump instruction (and delay slot
    execution)
# Expected: r03=0x5D5F
# Expected: r05=0xCAFF
# Expected: r04=0x1D5A
# Expected: r02=0xCAFE
# Expected: r06=0x0001
# Expected: r07=0x1D5F
        .align 2
        zldi r5,0xCAFE
        zldi r2,0xCAFE
        zldi r4, 0x0FFF
    zldi r3,0xDEAD
        ori4 r5,r5,1
        and r4,r3
        zldi r3,0xBABE
        slli r4,0x0001
        zldi r6,0x001
        ori4 r7,r4,0x000F
        sra r3,r6
    halt 0
```

K.U - Test program examples .vmem

```
@00000000   af
@00000001   fc
@00000002   1e
@00000003   05
@00000004   af
@00000005   fc
@00000006   1e
@00000007   02
@00000008   ff
@00000009   f0
```

```
@0000000a  1f
@0000000b  04
@0000000c  ea
@0000000d  fd
@0000000e  1d
@0000000f  03
@00000010  51
@00000011  05
@00000012  03
@00000013  74
@00000014  ab
@00000015  fb
@00000016  1e
@00000017  03
@00000018  71
@00000019  84
@0000001a  00
@0000001b  f0
@0000001c  11
@0000001d  06
@0000001e  4f
@0000001f  07
@00000020  46
@00000021  73
@00000022  03
@00000023  e0
```