NTNU
Det skapende universitet

Arkitektur-beskrivelse for AHEAD

Lars Olav Refnin

Master i elektronikk
Oppgaven levert: Juni 2006
Hovedveileder: Kjetil Svarstad, IET

Norges teknisk-naturvitenskapelige universitet
Institutt for elektronikk og telekommunikasjon

Oppgavetekst

For å generalisere AHEAD-plattform skal det studeres såkalte
”Architecture Description Languages”, ADL. Et prosjekt denne høsten har
undersøkt dette mulige AADL (AHEAD ADL), og kommet med en del forslag
til hvordan dette kan brukes for en tidlig generasjon AHEAD
klient-server funksjonalitet. Det er ønskelig å analysere dette mhp. å
utvide det til å kunne dekke nye generasjoner av AHEAD og også å analysere evt. spesifisere et
system forhvordan å utvikle og håndtere slike beskrivelser basert på de faktiske verktøy for FPGA-
utvikling somfinnes i dag. AADL sitt praktiske bruksområdet skal diskuteres opp i mot AHEAD sin
arkitektur. En presentasjon av grensesnittsmuligheter for moduler med tanke på
gjenbruk av moduler i AHEAD skal vises.

Oppgaven gitt: 23. januar 2006
Hovedveileder: Kjetil Svarstad, IET

Sammendrag

Rapporten er skrevet fordi AHEAD prosjektet ser behovet for et ADL til automatisk plassering av HW moduler og SW moduler innad på en FPGA. AHEAD er en videretuvikling av Amibesense, men inneholder ingen generell prosessor, men kun en FPGA i sin basestasjon.

AADL skal i stor grad ha den samme funksjonalitet i AHEAD som et ADL har for SW. Dette vil si at AADL skal beskrive en arkitektur av HW moduler og SW program sammen, heretter kalt system. Et problem med dette er at det ikke finnes noen ADL'er for HW og dette må derfor utvikles. ADL-språkene har et innebygd problem; En spesifikasjon kan aldri bli bedre enn den tid og kostnad som er lagt ned i spesifikajsonen.

Rapide er det språket som i dag eksistere som er nærmest den funksjonalitet som er ønsket i AADL. Rapide er ikke et rent ADL, men kan også brukes til simulering og verifisering på oppførselsnivå.

Kompilatoren til AADL trenger ikke å ha fri grammatikk og semantikk. Nøkkelord og topp-ned kompilasjon er derfor å foretrekke.

FPGA-verktøyene som eksisterer i dag støtter dynamisk rekonfigurering. Men de setter krav til klokke, benytter buss makroer og at man bruker det verktøy som Xilinx har utviklet.

AADL setter krav til AHEAD-arkitekturen, designeren og modulene som skal benyttes. Kravene AADL setter er en samling av HW og SW sine funksjonelle krav til AHEAD. Alle filer som blir brukt i AHEAD må være eksekverbare uten behov for mer behandling. Grensesnittet AADL legger opp til er ICA, som krever at modulene tilegenet AHEAD er designet for dette grensesnittet. Kommunikasjonsprotokollene til AHEAD må være ferdig definert til bruk i AADL da det valgte abstraksjonsnivået krever dette. Definisjonen av en kommu-

nikasjonsprotokoll må skje utenfor AADL. Antall grensesnitt og kommunikasjonsprotokoller en modul har vil ikke ha noen innvirkning for hvordan AHEAD tolker AADL koden.

Målet er at AADL koden holdes enkel med få innebygde funksjoner og med et topp-ned komiplator for øyet.

Forord

Rapporten inneholder en studie rundt AADL, AHEAD architecture describing language. Studien er lagt til rette for å se om AADL kan være til hjelp i en tidlig del av AHEAD-prosjektet.

Figurer vil inneholde engelske uttrykk da orginal kildene ofte er skrevet på engelsk. Det har vært problematisk å finne litteratur, materiale til denne studien da AADL er et delvis HW ADL-språk.

Takker gruppen rundt AHEAD og da spesielt Kjetil Svarstad for de innspill og den hjelp de har kommet med underveis.

Innhold

Sammendrag i

Forord iii

1 Introduksjon 1

1.1 AmbieSense 1
1.2 AHEAD 2
1.2.1 Verktøy i AHEAD – AADL 2
1.3 Ordforklaringer og begrepspesifikasjon 4

2 Arkitekturbeskrivende språk, ADL 5

2.1 Målsettingen for et ADL 5
2.2 ADL problemet 6
2.2.1 Detaljnivå i ADL 8

3 Rapide 9

3.1 Generellt om Rapide 9
3.2 Grensesnitt 11
3.3 Eksekvering av arkitektur 11

4 Kompilatorteknikk 13

4.1 Oppbyggning av kompilator 13
4.2 Analysen av kilden 14
4.2.1 Lexical 14
4.2.2 Syntaks 15
4.2.3 Semantisk 15
4.2.4 Blokk 15

5 FPGA-verktøy 17

5.1 Modulbasert rekonfigurasjon 17

5.2 Diffrensiert rekonfigurasjon 18

6 AADL 19

6.1 AHEAD arkitekturen 19

6.2 Krav til AADL 20

6.2.1 Krav og tilbud 21

6.2.2 AADL filoppbyggning 22

6.2.3 Autogenerering av AADL 23

6.2.4 Klokke og reset 23

6.2.5 Hierarkiskesystemer 24

6.2.6 FPGA – areal og rekonfigurering 25

6.2.7 HW/SW Partisjonering 26

6.2.8 Parametrisering av moduler 26

6.2.9 SW krav til HW 27

6.3 Filer brukt i AHEAD 28

6.4 Grensesnitt i AADL 28

6.4.1 OCA 29

6.4.2 ICA 30

6.4.3 SW grensesnitt mot HW 31

6.5 Kommunikasjonsprotokoller 32

6.5.1 SW synkron - asynkron kommunikasjon 33

7 Kodeforklaring og AADL eksempler 35

7.1 AADL kode eksempler 35

7.1.1 Mpeg-4 dekoder 35

7.1.2 SW eksempel 38

7.2 Kodeforklaring 39

7.2.1 Kodeoppdeling 39

7.2.2 Plassering i HW 40

7.2.3 HW og SW i samme AADL 40

7.2.4 Antall grensesnitt 41

7.3 AADL begrensninger 41

8 Konklusjon 43

8.1 Fremtidig arbeid 43

A AADL kode forklaring 49

Figurer

1.1 AmbieSense Arkitekturoversikt [1] 1
1.2 AHEAD arkitekturoversikt 3

3.1 Rapides oppbyggning.[2] 10

4.1 Virkemåten til en kompilator 13

5.1 Buss makro.[3] 17

6.1 AHEAD-arkitektur V3.0 19
6.2 Filoppbyggning til AADL 22
6.3 Hierarkisk blokksystem 25
6.4 OCA[2] 29
6.5 ICA[2] 30
6.6 SW HW standarisert grensesnitt.[4] 31
6.7 Grensesnitt eksempel 32

7.1 Barco Silex MPEG4 Dekoder.[5] 36
7.2 Barco Silex MPEG4 Dekoder, svart boks.[5] 36

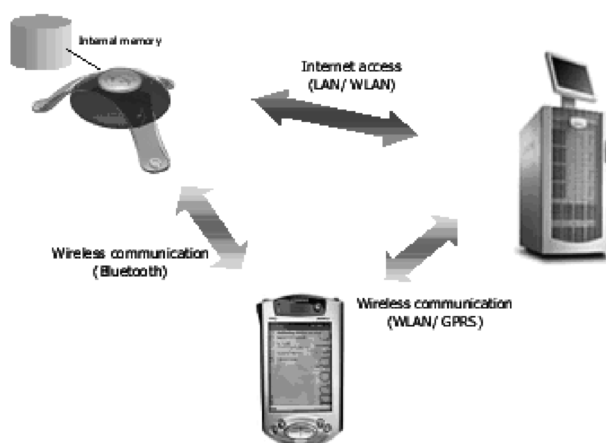
Kapittel 1

Introduksjon

Målet med denne oppgaven er å lage en del av verktøykjeden som skal brukes i prosjektet AHEAD. AHEAD er en videreutvikling av AmbieSense prosjektet.

1.1 AmbieSense

AmbieSense er i korte ordelag ett system for små plattformer som PDA'er, mobiler og lignende systemer med liten databehandlingskapasitet skal kunne formidle trådløst et problem til en kraftigere enhet med større databehandlingskapasitet og få et svar tilbake som ligger utenfor den orginale enhetens prosesseringskapasitet.[6][1]



Figur 1.1: AmbieSense Arkitekturoversikt [1]

Figur 1.1 viser en systemskisse over AmbieSense. Her ser man at alle enheter kan kommunisere med hverandre, men hovedpoenget er at basestasjonen kan ta seg av både kommunikasjon mot internett og det lille systemet. Basestasjonen har ofte en større kommunikasjonskapasitet enn den håndholdte enheten.

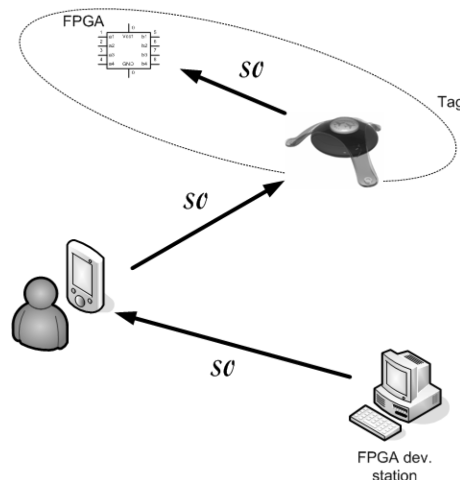
1.2 AHEAD

AHEAD er en videreutvikling av AmbieSense prosjektet. Figur 1.2 viser i stor grad hvordan AHEAD er bygget opp; Forskjellen mellom AHEAD og AmbieSense er at AHEAD ikke har en tradisjonell prosessor i basestasjonen, men en FPGA. Dette gjør at AHEAD ikke har like store prosesseringsegenskaper som den tradisjonelle basestasjonen i AmbieSense, men ved spesielle oppgaver vil AHEAD ha en mulighet for større datagjennomstrømmning. Et eksempel på dette er datakompresjon som kan gjøres raskere på en FPGA enn på en vanlig generell prosessor [7]. IP, *intellectual property*, er en softwarebeskrivelse av en Hardware-krets, HW, som tolkes og gjøres om til en krets i en FPGA. Ulempen med AHEAD er at det håndholdte systemet må inneholde en IP for den bestemte oppgaven som skal utføres og denne må lastes opp til basestasjonen, tagen. Hvis det håndholdte systemet har behov for å laste opp SW må det i AHEAD også laste opp en HW plattform denne SW kan ekskveres på. Et mål i AHEAD er at brukeren ikke vil trenge å bry seg om det er HW, SW eller systemer av disse som brukes. Fordelen er i AHEAD mot Ambiesense er at SW og HW kan brukes om hverandre og til de oppgaver hvor dette er formålstjenelig.

1.2.1 Verktøy i AHEAD – AADL

AADL er ment å bli et verktøy til bruk i AHEAD, og hovedhensikten er bruk i AHEAD hvor flere systemer må kommunisere med hverandre eller flere brukere benytter seg av basestasjonen/tag samtidig. AADL vil også være et verktøy for utviklere ved gjenbruk av moduler, ved at AADL gir en formell beskrivelse som skal være lettfattelig og oppgi nok informasjon slik at gjenbruk blir mulig. Figur 1.2 viser

en oversikt fra utviklingsplattform helt frem til FPGA gjennom både bruker og tag. AADL vil være en beskrivelse over IP'er og deres oppførsel og kommunikasjonsbehov både innad på FPGA'en og ut mot resten av verden.



Figur 1.2: AHEAD arkitekturoversikt

Ingar Hauge har skrevet en oversikt over ADL'er og analysert funksjonaliteten til de forskjellige språkene opp i mot et tenkt antall case til bruk i AHEAD. Ut i fra denne rapporten menes Colif og Unicon for å være de beste[8]. Både Unicon og Colif er "døde" prosjekter hvor det ikke lenger er mulig å få kontakt med personer som har jobbet med prosjektet eller få tak i relevant informasjon eller verktøy. Så disse språkene er ikke eller i liten grad vektlagt i denne rapporten.

De mer tradisjonelle ADL'ene er i dag stort sett døde. Med unntak av noen artikler og manualer er verktøyene som disse språkene hadde til rådighet ikke lenger å få tak i og de siste artiklene er ofte skrevet i slutten av 1990 årene.

1.3 Ordforklaringer og begrepspesifikasjon

Noen begreper brukes i både HW og SW sammenheng, men har ikke samme betydning for begge fagmiljøer. For å unngå misforståelser i denne rapporten blir det derfor spesifisert hvilken tolkning som skal brukes der det kan oppstå tvil.

ADL Arkitektur beskrivende språk/Architecture describing language.

AHEAD Ambient Hardware, Embedded Architecture on Demand.

AADL AHEAD ADL. Et ADL bergnet og utviklet spesielt for AHEAD.

Designer Vil bli brukt både om en SW programmerer og en HW designer.

FPGA Field Programmable Gate Array, programmerbar elektronikk.

ICA Interface Connection Architecture.

OCA Object Connection Architecture.

Tag Tag er basestasjonen i AHEAD arkitekturen.

Begrepspesifikasjon

System

System brukes både om HW arkitekturen og HW SW arkitekturen sammen.

Synkron - Asynkron kommunikasjon

Synkrone- og asynkrone design er begreper som ikke representerer det samme i HW som i SW. I denne rapporten menes HW synkrone - asynkrone systemer med mindre annet er spesifisert.

Kapittel 2

Arkitekturbeskrivende språk, ADL

Et ADL er et verktøy til bruk i utviklingen av et system, hvor et systemet kan være en samling av SW programmer, delprogrammer eller moduler som til sammen blir et program. ADL brukes til å beskrive systemet slik at brukerne eller designeren vet hvilke deler som utfører hvilke oppgaver og hvordan de skal knyttes sammen. Det typiske ADL'et har kun vært ment til bruk på SW. Noen unntak har dukket opp etter hvert som kompleksiteten i HW-systemene har vokst. Rapide er et ADL'er som kan modellere HW, SW, HW/SW systemer og er laget med tanke på blant annet co-design[2].

2.1 Målsettingen for et ADL

Målet til et hvert ADL er å beskrive systemet godt nok til at brukeren raskt får et overblikk slik at man fort kan gjenbruke eller konstruere modulen ut fra modellen som er skissert i ADL'et. Et godt ADL kan i stor grad sammelignes med en spesifikasjon da ADL'et i stor grad vil si det samme, men et ADL er formalisert, strukturert og har fastsatte regler. Med dette menes at ADL konstrueres opp mot en låst struktur og all informasjon formidles på innenfor denne strukturen. En konvensjonell SW spesifikasjon, som ikke er satt opp som et ADL, følger i stor grad tre kriterier[9]:

Komplett Dette vil si at brukeren kun får den informasjonen som

blir dømt til å være tilstrekkelig og/eller kun den informasjonen som man blir tillat å få.

Statisk Inneholder få oppgraderinger etter at den først er skrevet selv om den originale funksjonaliteten er endret.

Homogen Samme spesifikasjons struktur blir brukt på alle moduler uavhengig av funksjon eller grensesnitt.

Dette siste kravet fører til at en spesifikasjonstype kan være ypperlig for en modul, men være helt ubrukelig for en annen. Derfor må en spesifikasjon for et system alltid være heterogen for å kunne inneholde all den informasjonen som kan tenkes være nyttig.

Det eksisterer tre kriterier som bør følges for at en modul skal ansees som gjenbrukbar[10]:

1. Gjennomsiktig avgrensning.
2. Komplett beskrivelse av grensesnittet.
3. Alle modulene må designes ensartet med tanke på gjenbruk.

Disse tre kriteriene er satt fra en programmerers ståsted og ikke fra en formell metode vinkling. Alle disse tre kriteriene er mulig å følge, men i en del tilfeller vil de være upraktiske eller lite funksjonelle. En gjennomsiktig avgrensning vil si at en ikke setter modulen, programmet eller delprogrammet i en svart boks, men lar bruker få tilgang til selve oppførselen. En komplett beskrivelse av grensesnittet er nødvendig for å kunne kommunisere korrekt og tolke informasjonen som kommer ut. En ensartet design kan være vanskelig å gjennomføre da de forskjellige delene av det totale systemet sannsynligvis vil ha et stort spenn i oppgaver og oppførsel. Dette vil si at det ikke alltid er best å designe med tanke på gjenbruk.

2.2 ADL problemet

En spesifikasjon vil alltid være utilstrekkelig for brukeren/programmeren da man aldri kan forutse alle mulige problem. Informasjonen som

trengs for å gjenbruke en modul er ikke alltid den rent oppførselbeskrivende, men må også inneholde strukturelle egenskaper som beskriver samhandlingen med andre moduler og grensesnitt.[9]

Homogen – heterogen problemet som nevnt i forrige avsnitt oppstår også i et ADL. All informasjon i et ADL er formalisert og man vil få problemer med å uttrykke informasjon som ikke ligger i språkets natur. Å lage et ADL som skal være heterogent vil være føre til ADL'er raskt blir stort og fort kan bli uoversiktelig. Fordelen med en designer skrevet spesifikasjon er selv om denne er formalisert så kan man alltid bryte reglene da en spesifikasjon er ment som informasjon til et menneske, enten en SW programmerer, HW designer eller bruker. Spesifikasjonen vil da fremdeles være leslig og være mer mer nyttig da mer informasjon vil være tilgjenglig. Men et slik måte å utnytte et ADL på vil være utelukket da det alltid skal være mulig for en maskin og behandle informasjonen.

I rapporten "A Surey of Architecture Description Languages [11] " refereres det til to lister over krav for hva ett ADL må inneholde som f.eks. modul abstraksjon og type sjekking. Listene inneholder krav som er delvis selvmotsigelige. Noe som indikerer at et ADL er noe som ikke er klart definert og at man har forskjellig oppfatning av hva som kreves og skal tilbys av et ADL.

En hypotese som er fremsatt av Mary Shaw[12]:

> Presisjonen i spesifikasjonen vil være en følge av en aveiing mellom kostnad og fordelene av en mer detaljert spesifikasjon.

Dette vil si at en spesifikasjon aldri vil bli perfekt, men man kun kan strebe mot godt nok for den gruppen som man har sett seg som mål. Et ADL vil også følge denne hypotesen da et ADL også vil lide under en aveiing mellom kostnad og detaljnivå. Hypotesen viser at det å beskrive mange type moduler, som f.eks. SW, HW, minne, prosessor eller grafisk demo, innenfor samme ramme er vanskelig og til dels lite nyttig pga at et ADL må være stort for å ikke være homogent. Et stort ADL med mange muligheter vil være lite nyttig da sjansen for å gjøre feil blir høyere og læringskurven, for noe som

kun skal være et verktøy, kan bli et hinder.

2.2.1 Detaljnivå i ADL

Et ADL er verktøy for å hjelpe til i en prosess som skal se etter en mulig optimal systemutnyttelsen. En optimal systemutnyttelse kan være alt fra kortest eksekveringstid for systemet til et system som krever minst mulig resursser å eksekvere. ADL som et verktøy skal hjelpe med å sette opp en oversiktelig arkitekturbeskrivelse og på denne måten være med på å korte utviklingstiden. Et ADL kan lages til alle detaljnivåer. Utforsking på pinne til pinne, klokkesykel til klokkesykel og/eller funksjon til funksjon vil bli for detaljert og heller ikke hensiktsmessig. Dette vil ta for lang tid, vil ha for skarp læringskurve og blir for vanskelig i designfasen. Et ADL skal ikke inneholde funksjonalitet og av denne grunn vil man ikke være intressert i å legge ned mye arbeid i ADLdelen av et prosjekt. En måte og sørge for dette på er å sørge for et høyt abstraksjonsnivå på arkitektur- og modellnivå [13].

Et ADL kan inneholde alle detaljnivåer, men de laveste nivåene vil være til liten nytte. Ved å sammenligne kriteriene satt i avsnitt 2.1 fra [10] opp mot hypotesen som er fremsatt i avsnitt 2.2 fra [12] vil man komme frem til at det er flere abstraksjonsnivåer som er hensiktsmessig. Hvilket abstraksjonsnivå som er hensiktsmessig avhenger av teknologi og det bruksfeltet ADL'en er utviklet for.

Kapittel 3

Rapide

Rapide er det ADL-språket som kommer nærmest den ønskelige funksjonalitet i AADL.

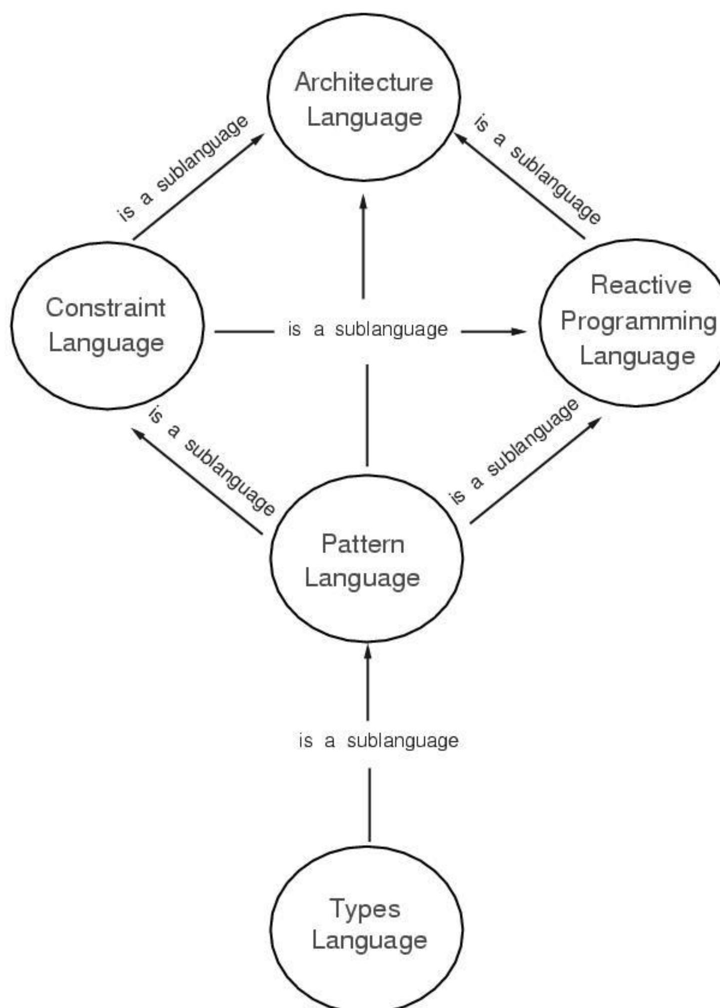
3.1 Generellt om Rapide

Rapide er ett event basert, objektorientert programmeringsspråk. Med event menes endringer, som for eksempel en klokke som endres eller en SW tilstandsmaskin som som endrer tilstand. En større arkitektur med mange moduler fremsetter et krav om parallel behandling av data og Rapide er konstruert for dette. Rapide brukes for å sjekke at den valgte arikitekturen gir ønsket respons gjennom simuleringer og er beregnet for prototyping av en arkitektur.[14]

Rapide skiller seg fra en del tradisjonelle ADL'er gjennom at det ikke bare beskriver funksjonalitet, men også kan simulere, bekrefte og verifisere at ønsket funksjonalitet eksistere i det tenkte arkitekturen. Rapide er av denne grunn ikke et ekte ADL.[14]

En typisk Rapide arkitektur består av modulenes grensesnitt, et sett med regler for direkte kommunikasjon mellom modulene og et sett med formelle regler som avgjør hvilken kommunikasjon som lovlig eller ulovlig. Kommunikasjonen være synkron eller asynkrone mellom modulene.[14]

Dynamisk arkitektur kan bli beskrevet og simulert på lik linje med statiske arkitekturer i Rapide. Dette fordi Rapide kan simulere med et varierende antall moduler og kommunikasjonskanaler under



Figur 3.1: Rapides oppbyggning.[2]

en enkelt simulering.[14]

Rapide skille seg fra andre programmeringsspråk og ADL'er fordi det er satt sammen av fem hoveddeler. Disse delene er underspråk av Rapide, se figur 3.1 på side 10. Underspråkene er kompatible mot hverandre grunnet deling av krav som for eksempel scope, navneregler og eksekveringsmodell.[2][14]

Oppdelingen av Rapide til fem deler gjør at hver enkelt del kan studeres og endre hver for seg. Endringene i en enkelt del har stor grad av frihet før de andre delene må endres i samsvar. [14]

3.2 Grensesnitt

Rapide har et spesielt grensesnitt som viser modulens krav til andre moduler i tillegg til at det også viser informasjon om hva modulen selv tilbyr av funksjoner og data.[14][2]

Både ICA og OCA som blir forklart i kapitell 6.4 blir støttet av Rapide. ICA er det grensesnittet som blir fremhevet for Rapide. Denne grensesnitt formen har innebygd støtte i Rapide.[2]

Grensesnittene kan også bli testet før selve modulen er, og dette gjøres ved at grensesnittet blir programmert løsrevet fra selve modulen.[2]

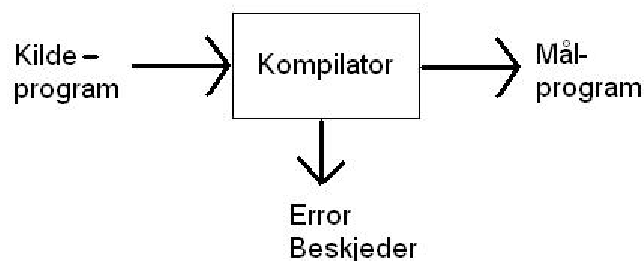
3.3 Eksekvering av arkitektur

Simuleringen av eksekverbar Rapide kode vil vise både tidspunkt og hvilke hendelser som utløser andre hendelser. Selvstendige hendelser blir vist i samme simulering. [2]

Kapittel 4

Kompilatorteknikk

En kompilator har en grunnfunksjon; Det er å oversette fra et språk til et annet. Figur 4.1 viser en oversiktlig virkemåte for en kompilator. En viktig del av prosessen i kompileringen er å gi brukeren tilbakemeling om eventuelle feil som har oppstått under veis. [15]



Figur 4.1: Virkemåten til en kompilator

Uttrykket kompilator er ett samle begrep som brukes om alle oversettere uavhengig av kompleksitet og funksjonalitet som tilbyes av kildespråket og uavhengig av hva målspråket er.

4.1 Oppbyggning av kompilator

De fleste kompilatorer er bygd opp på et to-delt prinsipp; analyse og syntese. Analysedelen deler opp kilden og bygger opp et mellom nivå som representerer kilden. Syntese delen ser på mellom nivået og lager målprogrammet. [15]

Målprogrammet kan være alt fra maskinspråk til et annet programmeringsspråk. Ett eksempel på det siste er et program og kom-

pilator som heter Formality og brukes i stor grad for å oversette mellom programmeringsspråkene Verilog og VHDL.

4.2 Analysen av kilden

Analysedelen av en kompilator er tredelt bestående av en lexical, syntaktisk og semantisk del. Alle tre har sine oppgaver, men alle tre eksisterer ikke i alle kompilatorer.

4.2.1 Lexical

Lexical metoden er en lineær analyse kompilator som leser kildespråket fra topp til bunn[15]. Hovedoppgaven til en en slik analysator er å oversette uttrykk i kildekoden og oversette disse til en sekvens av tokens, iditentifikasjoner, konstanter, strenger og seperatorer[16].

David Blasband har delt opp Lexical kompilatorer opp i fire nivåer hvor nivå 0 er ingen evne til å se bakover og if-setninger ikke vil kunne brukes. Nivå 1 vil tillate enkle rekursive metoder i kompilatoren og vil derfor kunne tillate f.eks. if-setninger. Det høyeste nivået er nivå 3 hvor man til enhver tid kan se hva som har skjedd og hvor; Dette er det ingen som har laget enda. [17]

Nøkkelord

Nøkkelord også kalt "reserved". I en lexical analysator deler opp kildekoden i flere småbiter kalt tokens. Tokens er den minste sammensatte samling av tegn som former et argument. Nøkkelord blir brukt for å reservere konstruksjoner, hvor en type konstruksjon kan være en if setning. Tokens blir av den lexicale analysatoren sammenlignet med nøkkelordene og setter så igang den assosierte konstruksjonene. En fordel med nøkkelord er at det ikke blir mulig å forveksle en token med et nøkkelord. Dette vil si at enhver gang et token passer med et nøkkelord vil den assosierte konstruksjonen settes igang. Følgen er at nøkkelord kun kan brukes når de assosierte konstruksjoner er ønsket. [15]

Viktige karakterer som "end-of-line" og mellomrom er en del av nøkkelordene. Disse blir gjenkjent og reservert på lignende måte som resten av nøkkelordene.

4.2.2 Syntaks

Syntaks analyse av blir ofte kalt *parsing*. Parsing er en form for hierarkisk analyse av kilden. Hierarkiet brukes til å bygge opp grupper som igjen settes sammen til et parse tre som kan syntetiseres. [15]

Skillet mellom Lexical kompilatorer og syntaktiske kan være vanskelig til tider. Syntaktiske språk tillater i større grad rekursive formuleringer i kilde språket enn det rent Lexicale kompilatorer gjør.

4.2.3 Semantisk

En semantisk kompilator er en videreføring av en syntaktisk kompilator. Det hierarkiske treet som ble bygget opp av den syntaktiske kompilatoren blir gjennomgått av den semantiske delen etter feil; først og fremst etter typefeil. Kildespråkets spesifikasjoner setter hvilke typer som kan refereres til hverandre og brukeren vil få beskjed fra dette nivået i kompilatoren hvis en feil oppstår eller blir funnet.[15]

4.2.4 Blokk

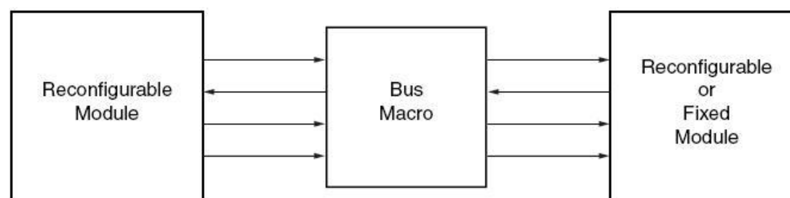
Med blokk menes en samling av argumenter ved hjelp av en konstruksjon eller funksjon. Blokk kan implementeres i en kompilator ved hjelp av en Stack. I det gjeldende programmerinsspråket finnes det flere måter å implentere enn blokk på. Både LaTeX 2_εog Algol bruker en begin end struktur. Problemet som kan oppstå ved bruk av blokk er at innholdet kun er deklarert på innsiden og følger blokkens scope.[15]

Kapittel 5

FPGA-verktøy

De verktøy som er mest andvendlige i AHEAD vil være en delvis rekonfigurasjon av FPGA'en mens den er i drift. Og det finnes i dag kun to produsenter, Atmel og Xilinx, som støtter en delvis og dynamisk rekonfigurasjon av en FPGA[18]. AHEAD er pr. idag basert på en Xilinx FPGA. Xilinx presenterer to metoder for en delvis rekonfigurasjon; Modul basert- , her kun HW, og diffrensiert rekonfigurasjon.

5.1 Modulbasert rekonfigurasjon



Figur 5.1: Buss makro.[3]

Ved bruk av modulbasert rekonfigurasjon må HW'en være oppdelt i klare moduler. Moduler som har behov for kommunikasjon med andre moduler få en buss makro i de øyeblikkene rekonfigurasjonen er i gang. Denne bussmakroen kan ikke være i bruk for den enkelte modul i det modulen rekonfigureres.[3]

Klokker bør distribueres globalt over FPGA'en ved design for delvis rekonfigurasjon. Dette for å redusere sjansen for en konflikt mellom klokker innad på FPGA'en.[3]

Dette vil si at et system kan holde alle moduler i gang untatt den modul som rekonfigureres i øyeblikket. I AHEAD vil få endriger være kjente og tiden som brukes til å rekonfigurere den enkelte modul vil være ukjent. All data som blir sendt på det tidspunkt FPGA'en er under rekonfigurasjon vil derfor kunne være korrupt.

5.2 Diffrensiert rekonfigurasjon

Ved programmering av en FPGA er det en bitfil som står for programmeringen. Diffrensiert rekonfigurasjon tar utgangspunkt i den orginale bitfilen og sammenligner endringene med den orginale filen. Forskjellen på disse to filene blir deretter skrevet til en bit fil som nå kan programmere FPGA'en til det nye systemet. Fordelen med dette er at selve konfigurasjonen nå vil være mye raskere grunnet at den delen som skal konfigureres er betraktelig mindre. Men selve konfigurasjonen har ingen forskjell fra en konfigurasjon som konfigurerer hele systemet. [3]

Ulempen med denne metoden å rekonfigurere en FPGA på er at kun små deler av logikk kan endres. Ved større endringer så er modulbasert rekonfigurasjon påkrevd.[3]

AHEAD bruker HW moduler og vil ofte kreve en stor endring på FPGA'en. Dette fører til at diffrensiert rekonfigurasjon ikke vil kunne brukes da kravet om at kun lite logikk kan endres.

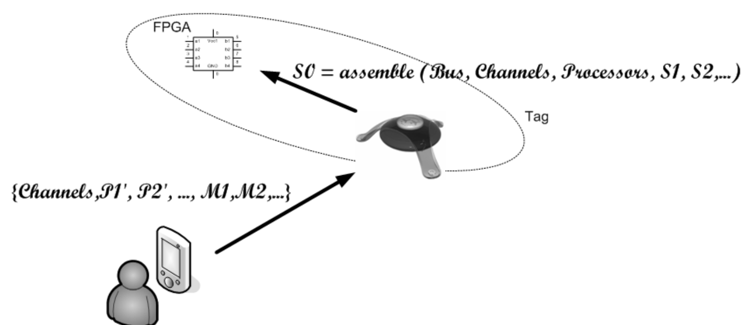
Kapittel 6

AADL

AADL skal brukes til å vise sammenhengen mellom moduler som skal lastes inn på en FPGA; FPGA'en sitter i tagen til AHEAD prosjektet som er den utførdende delen av AHEAD arkitekturen. Modulene kan være enten SW eller HW. AADL vil legge mest vekt på HW modulene da FPGA'en i AHEAD ikke vil kunne kjøre rene SW moduler uten en HW modul som plattform.

AADL skal ikke være et rent ADL-språk, men skal ha en underliggende funksjonalitet til grunn. Rapide er det ADL språket som kommer nærmest i funksjonalitet da det også beskriver HW.

6.1 AHEAD arkitekturen



Figur 6.1: AHEAD-arkitektur V3.0

Figur 6.1 viser den tenkte endelige versjonen av AHEAD. Her har sluttbrukeren kun en håndholdt enhet og en tag å forhole seg til.

AADL setter krav ut i fra denne siste versjon av AHEAD og ikke de versjoner som ligger til grunn i utvikling som figur 1.2 på side 3. Her er utviklingsstasjonen tatt ut av arkitekturen og AHEAD må behandle AADL uten ekstern hjelp.

AHEAD har ikke bestemt hvordan tagen skal fungere teknisk eller hvordan kommunikasjonen skal foregå internt i tag eller mellom tag og håndholdt enhet.

6.2 Krav til AADL

AADL skal være et arkitekturbeskrivende språk, ADL, og dette vil si at det språket ikke skal inneholde funksjonalitet, men kun beskrive funksjonalitet i moduler og kommunikasjonen i systemet.

Må kunne behandle HW <–> HW.
Må kunne behandle SW <–> SW.
Må kunne behandle SW <–> HW.
Alle tre overstående krav må kunne behandles i samme arkitektur.

Tabell 6.1: Hovedkrav til kommunikasjon i AADL.

Kravene som er nevnt i tabell 6.1 er utdypet og spesifisert nærmere til delkrav som må oppfylles for at AADL skal få den ønskede funksjonalitet. Delkravene for AADL er oppdelt i tre kategorier; HW, SW og felles krav for begge kategorier. Tabell 6.2 på side 21 viser delkravene. Delkravene er bygd på kravene for gjenbruk av IP'er, som er hentet fra Michael Keatings Reuse Methodology Manual[19]. Gjenbruk er en viktig for AHEAD prosjektet da et av målene er automatisk partisjonering og plassering av moduler på en FPGA. AHEAD vil ikke ha et gjenbruk på samme måte som i et vanlig SW eller HW prosjekt hvor gamle SW moduler skal brukes sammen med nye SW for å danne en ny funksjonalitet eller HW prosjekt hvor teknologien endres. Gjenbruk i AHEAD vil si at en modul skal kunne brukes på samme FPGA og grensesnittet skal være det samme i forskjellige systemer som kan settes opp i AHEAD. Flytting av moduler innad på FPGA'en uten at dette får innvirkning på funksjonalitet er et viktig kriterie. AADL skal være et verktøy til bruk i den automatiske

partisjonering og konfigurasjonen som gjøres i AHEAD.

For HW	For SW	Felles krav HW SW
Fysiske begrensinger Plassering på FPGA Synkrone/Asynkrone - Grensesnitt	Krav til ytelsen på HW Arkitektur Minne	Total ytelse Partisjonering

Tabell 6.2: Delkrav til AADL

Det er ikke satt at AHEAD skal kun inneholde synkrone eller asynkrone design og derfor vil er det viktig at AADL bygges opp til å håndtere begge designmetoder og tillatter begge former for design. Et design system kan i noen tilfeller inneholde både asynkrone og synkrone design og det vil derfor være viktig at begge to kan beskrives i samme arkitektur. Som en følge av dette vil noen moduler ha både et asynkront grensesnitt og et synkront grensesnitt; Muligheten for å beskrive dette i AADL må være til stedet.

AADL skal ikke legges til grunn for verifisering av at systemet har korrekt funksjonalitet, men kun være et hjelpemiddel i partisjoneringen av systemet og verfisieringen av at alle moduler har tilgang på alle de ressurser som den enkelte modul krever.

6.2.1 Krav og tilbud

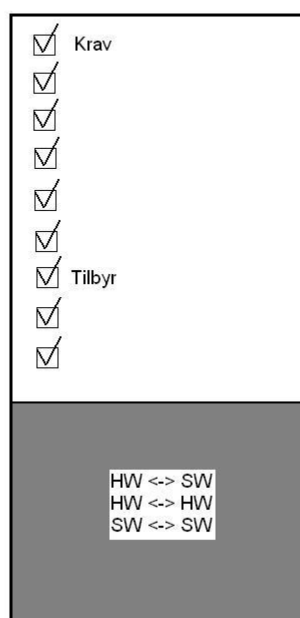
AADL skal presentere for designeren og systemet hva den enkelte modul tilbyr av funksjonalitet, og hva den krever for å fungere. Dette vil si at beskrivelsen av tilbud og krav vil ligge på et så høyt som mulig abstraksjonsnivå som f.eks. antall I/O'er en HW modul trenger eller hvor stort minne en SW modul vil ha behov for.

Navnerom

Et problem med krav og tilbud metoden er at en standarisert beskrivelse av en modul i AADL ikke kan forutse alle tilbud og krav som det vil kunne oppstå behov for å beskrive. Dette vil si at ikke alle muligheter kan ligge i faste parametre i AADL, men må navngies spesielt av designeren. Eksempler på dette er instruksjonsett, databusser og lignende. En fordel med navnerom er at synkrone og

asynkrone grensesnitt nå kan beskrives på samme måte. Problemet med navnerom er at den enkelte designer kan sette forskjellige navn på samme funksjonalitet og dette vil da bli behandlet som to forskjellige funksjoner i en krav og tilbud metodikk. Navnerommet bør derfor spesifiseres og standariseres, i hvert fall innenfor samme prosjekt.

6.2.2 AADL filoppbyggning



Figur 6.2: Filoppbyggning til AADL

Figur 6.2 på side 22 viser en grafisk oversikt over hvordan en AADL fil er bygget opp. Den øverste delen av figuren viser en sjekk liste som blant annet inneholder tilbud krav spesifikasjon for den enkelte modul. I denne sjekklisten vil det også ligge informasjon om hva som kreves av AHEAD tagen. Dette blir målt opp mot hva som hele systemet krever og en tilbakemelding kan bli gitt hvis ikke nok resurrser er ledig. Areal, antall I/O'er og spesifike HW blokker på FPGA'en som RAM blokker er alle krav som kan rettes spesifikt til AHEAD som et genrellt system. Disse kravene rettes ikke til den arkitekturen som er i ferd med å bygges opp men til selve plattformen.

SW har ikke lignende krav til plattformen[1], men setter krav til HW modulene i systemet og kan da ha innvirkning på plasseringen av HW, og krav til plattformen, AHEAD, gjennom dette.

6.2.3 Autogenerering av AADL

Autogenerering av AADL utifra de orginale programfilene i VHDL, Verilog og/eller et eller flere SW beskrivende som f.eks. Java er en målsetning for AADL. Deler av AADL koden vil kunne bli autogeneret: Slikt som plassering av porter og om det er en SW modul eller en HW modul. Dette er vist som det grå feltet i figur 6.2 på side 22. Et eksempel på autogenerering er HW modulenes grensesnitt i AADL.

I både SW og HW kildekodene vil det være en fordel om man kan skrive inn AADL kode som vil være med i en autogenerering av AADL. På denne måten vil en designer lettere få kontroll på hva AADL filen beskriver ved endring av koden og kan rette AADL filen direkte. Et eksempel på ekstra kode i en kildekode er Javadoc. Javadoc inneholder ingen funksjonalitet, men kompileres til dokumentasjon for funksjonalitet for den gjeldende javafilen.

Hvordan modulene kommuniserer, som gjennom predefinerte protokoller, med omverdnen er vanskelig å gjennkjenne automatisk og vil kreve et kraftig analyseverktøy og det vil derfor være mest hensiktsmessig å la designeren ta seg av denne biten i AADL. Kapittel 6.5 på side 32 beskriver problemstillingen rundt kommunikasjonsprotokoller.

6.2.4 Klokke og reset

Ved design i HW brukes klokker og forsinkelse ved distribusjon av disse utover i et design kan være et problem. AADL må inneholde støtte eller ikke underbygge mulige løsninger for design som minimaliserer klokkeproblemet. En mulig løsning på klokkeproblematikken er et system hvor en klokke ligger lokalt i kun en modul og ikke distribueres over hele systemet. Dette får også den følge av at det kan legges flere klokker i systemet og man får da en egenskap

[1]At SW ikke krever det samme av plattformen som HW er en antagelse.

om at de forskjellige modulene kan kjøres på ulike klokkehastigheter. Klokker som kjøres lokalt i en modul kan skrues av når modulen ikke er i bruk og det ikke har oppstått et behov for å fjernet den fra FPGA'en. Flere lokale klokker vil kunne gi en differanse på moduler som skal kommunisere over et synkront grensesnitt. Derfor vil en global klokkefordeling være nødvendig. En HW klokkeport vil ikke bli sett på annerledes av AADL enn andre HW-porter. Fordeling av klokker vil bli opp til den enkelte designer og prosjekt.

Reset bør kunne kjøres både lokalt for den enkelte modul og globalt for hele systemet. Dette er rettet mot en versjon av AHEAD hvor flere brukere kan dele samme basestasjon og f.eks. en bruker har behov for å resette sin del av arkitekturen være det HW, SW eller begge deler. AADL vil ikke beskrive reset funksjonalitet, men la reset være en HW-port på linje med andre signaler. Problematikken rundt lokale og globale reset må løses av designerene for den enkelte modul eller system. AHEAD prosjektet kan legge opp til at alle moduler skal ha en lokal og en global reset og vil da i stor grad ha løst problematikken.

Hierarkiskruting

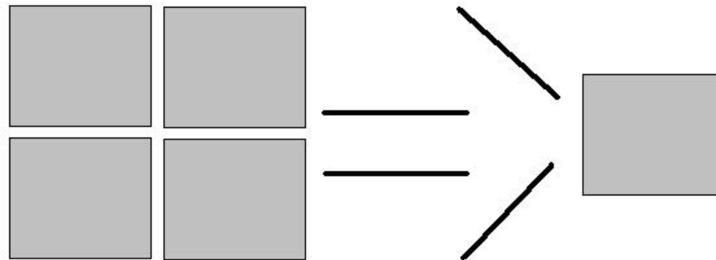
I et større system vil det ofte finnes signaler som vil gjelde for alle eller nesten alle undersystemer av moduler, eksempler på dette er klokke og reset. Noen signaler vil være globale og ha innvirkning på hele systemet og dets oppførsel. Signaler som kun gjelder enkelte undersystemer vil også finnes, for eksempel der hvor kun nettkontrolleren skal resettes.

AADL vil ikke diffrensiere mellom globale eller lokale signaler, men kun holde seg til et nivå. Dette gjøres for å legge til rette for en autmatisk generering av AADL ut fra kildekode og legge til rette for designeren skal bruke minst mulig tid på å skrive AADL.

6.2.5 Hierarkiskesystemer

I en arkitektur over et system vil det ofte finnes flere moduler som deler eller benytter seg av de samme funksjonalitetene eller ressur-

serene. Av denne grunn kan det være praktisk for en designer å ha et hierarkisk system hvor modulene kan beskrives i en blokk som et sammendrag.



Figur 6.3: Hierarkisk blokksystem

Figur 6.3 på side 25 viser at 4 blokker skal kunne beskrives som en. Dette vil si at den ytre blokken som representerer de andre blokkene vil fungere som et sammendrag. Poenget er å vise for både systemet og andre programmerer at disse blokkene fungerer som et system eller et delsystem. Hovedhensikten med et hierarkisksystem er å gjøre det lettere for en designer å finne i en stor arkitektur de blokkene som representerer en gitt funksjonalitet. For behandling inn til selve basestasjonen/tagen i AHEAD vil denne funksjonen ha liten eller ingen betydning da en maskin ikke vil ha samme problem med å strukturere informasjon på samme måte som et menneske.

I et hierarkisk system vil alltid nivået over være en abstraksjon av de under. Et problem som oppstår ved et slikt sammendrag er at ikke alt fra forrige nivå skal eller bør beskrives. AADL kan ikke akseptere et tap i et øvre nivå da alle detaljer er essensielle for en maskin tolkning. Derfor vil AADL ikke implentere et hierarkisk system.

6.2.6 FPGA – areal og rekonfigurering

Rekonfigurering på en FPGA kan gjøres på to måter; modul basert rekonfigurasjon eller forskjelles basert rekonfigurasjon, se kapittel 5. AHEAD prosjektet har ikke i skrivende stund bestemt seg for hvilken rekonfigurasjonsmetode som skal brukes eller om en egen metode skal utvikles.

Arealet på en FPGA er låst i et visst antall LUT, CLB og rutingsnettverk. Dette fører til at det arealet en modul opptar er viktig å minimere for å få plass til flest mulig moduler. I/O til og fra FPGA'en er en begrenset ressurs som en del moduler er helt avhengige av og da spesielt i de tilfeller hvor et skreddersydd grensesnitt er nødvendig. Skreddersydde moduler som er helt avhengig av en spesiell plassering på FPGA'en kan bli et stort problem. En mulig løsning på dette er at som kan flyttes på må være fleksible og ikke ha noen preferanse for plassering.

6.2.7 Hw/sw Partisjonering

Med HW/SW partisjonering menes her valget mellom å la en modul være i HW eller i SW. Keaton sier i sin bok[19] at dette valget i stor grad må gjøres manuelt av en designer og baseres i stor grad på tidligere erfaringer og en forståelse av de forskjellige fordelene og ulempene ved HW og SW.

Partisjonering vil i AHEAD legges i stor grad opp til at desginerne velger HW eller SW; Hvor HW kan bli det foretrukkne alternative på grunn av AHEAD sin arkitektur. I de tilfeller hvor en designer velger å lage en modul i både HW og SW skal AADL legge opp til en automatisk partisjonering hvor det mest hensiktsmessige på det gitte tidspunktet skal velges. Dette betyr at i et gitt tilfelle vil en modul være HW for å så skifte til en SW i en rekonfigurasjon av FPGA'en. Skiftet mellom HW og SW må utføres slik at bruker ikke merker et datatap eller et opphold av datastrøm der hvor dette brukes.

6.2.8 Parametrisering av moduler

Problemene som areal, minne og prosesseringskapasitet kan løses ved at de enkelte moduler inneholder parametre som avgjør fleksibilitet. Med fleksibilitet menes her at den enkelte modul kan kutte i sitt tilbud, flyttes innad på FPGA'en eller konverteres til SW eller HW hvis denne partisjoneringen oppfattes som bedre. Dette kan føre til at modulen må eksistere i både HW og SW innenfor samme beskrivelse i AADL. Ulempen med denne flesibiliteten er at det håndholdte

systemet må bruke mer minne for å inneholde både en SW beskrivelse og en HW beskrivelse av samme modul og i tillegg en versjon for hver parameter endring i både SW og HW. Hvis mange moduler blir dobbelt programmert vil fleksibiliteten til systemet bli høyt og man kan få mest mulig ut av de begrensede ressurser som en FPGA innehar. Muligheten modulen har med å kutte i sitt tilbud vil være for eksempel en minneenhet som da vil realisere mindre minne på FPGA'en eller en SW modul som vil tilby mindre funksjonalitet til sine omgivelser og på den måten bruke mindre minne og/eller ha behov for mindre prosesseringskapasitet.

6.2.9 Sw krav til HW

All SW har krav til HW. Disse kravene kan være variasjoner av minne, dataflyt og/eller prosessorkrapasitet. Dataflyt er her ment som kravet fra SW til HW som et krav om mengden data som kan behandles av gangen, enten i parallell eller serielt. Prosessorkapasitet måles i CPI, antall klokkesykluser pr instruksjon. CPI er vist i formel 6.1, hvor IC er antall instruksjoner som blir eksekvert.[20]

$$CPI = \frac{\text{CPU klokkesykluser for et program}}{\text{IC}} \tag{6.1}$$

CPI er ikke en god målestokk for å måle ulike prosessorer opp mot hverandre, men i AADL er det heller ikke et behov for slike målinger[20]. Instruksjonssett til prosessoren påvirker CPI og gir derfor en god pekepinn på om den aktuelle prosessoren er i stand til å eksekvere denne SW'en[20]. Dette fordi SW i AHEAD er ferdig kompilert med hensyn på prosessortype.

Sw styrt plassering av HW

Det kan senere oppstå situasjoner hvor SW krever tilgang til spesielle HW moduler eller krav til forsinkelse i HW i SW. Dette vil si at SW setter større krav til HW plassering enn det HW selv gjør.

6.3 Filer brukt i AHEAD

Filene som skal lastes inn på AHEAD tagen skal være eksekverbare. Med dette menes det at filene enten er bitfiler for å programmere FPGA'en eller at det er SW filer hvor det finnes HW til stedet på tagen som kan behandle disse. Dette vil si at filene ikke skal behandles av AHEAD før de lastes inn på tagen. Dette setter et krav til at alle moduler har alle konstanter satt i den eksekverbare koden og er derfor fullstendig konfigurert. Fleksibiliteten til systemet blir mindre når alle konstanter er satt og ingen endringer på konfigurasjonen blir tillatt på den enkelte modul.

Ved å tillate endringer av konstanter vil en større fleksibilitet kunne oppnåes, men dette vil sette krav til at AHEAD kan kompilere. Det vil settes krav til at AADL legger opp til et fornuftig valgsett av konstante i det totale systemet. Denne siste løsningen vil sette høye krav til den HW som ligger i tagen i AHEAD og den HW og SW som må ligge der konstant for å utføre den siste kompileringen og valgene som må taes. Sett fra det utgangspunkt som ligger i figur 6.1 på side 19 er løsningen med fleksible konstanter ugunstig på grunn av kravene til HW og SW og de ressursene disse vil oppta. Kapittel 6.2.7 på side 26 legger opp til en automatisk partisjonering. I de tilfeller hvor denne valgprosessen vil oppta for mye ressurser enten av design eller i AHEAD vil denne muligheten kunne fjernes.

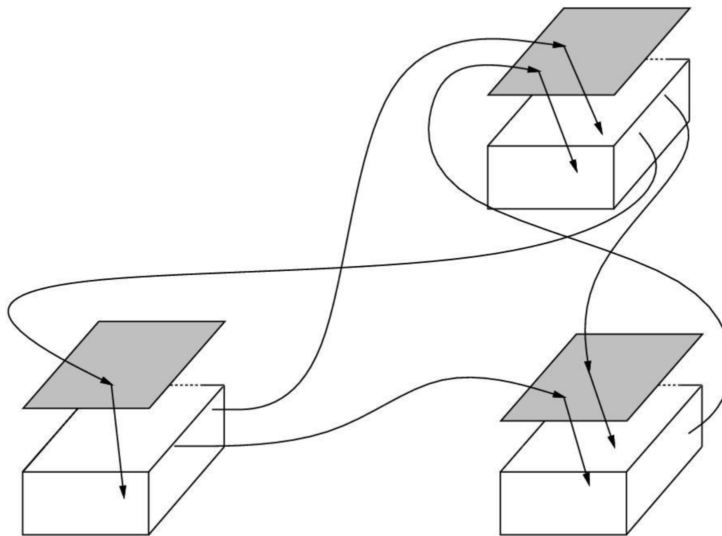
6.4 Grensesnitt i AADL

HW og SW har forskjellige krav til sine omgivelser. HW krever blant annet areal på FPGA'en, I/O og rutingsnettverk mens SW krever blant annet minne og prosesseringskapasitet. Dette fører til at AADL må vise sammenhengen mellom moduler uten å måtte vise de protokoller eller føringer som ligger mellom modulene på detaljnivå. AADL vil ikke beskrive noen busstandarder eller SW HW grensesnitt og lar det være opp til det enkelte designer, prosjekt å definere hvordan kommunikasjonen skal foregå. Kommunikasjonen mellom de forskjellige moduler må som følge av dette ha en mulighet for å bli

beskrevet abstrakt. Dataflyt mellom moduler er ikke en del av et grensesnitt og må derfor defineres utenfor AADL. Dette med tanke på at AADL først og fremst skal brukes til plassering i et system på en FPGA.

I tabell 6.1 på side 20 nevnes det noen krav til hva AADL skal inneholde av informasjon om kommunikasjonen. Begge grensesnitt typene ICA og OCA oppfyller kravene i denne tabellen.

6.4.1 OCA



Figur 6.4: OCA[2]

OCA beskriver hvordan moduler skal kunne kommunisere med hverandre. Denne formen for beskrivelse belager seg i mindre grad på et grensesnitt og beskriver mer direkte oppførselen/funksjonaliteten til selve modulen. Modul referanser er ikke bare til grensesnittet, men til funksjonalitet innad i en annen modul. En følge av OCA tenkemåte er at modulen ikke kan beskrives i et ADL før modulen er helt ferdig designet.[21]

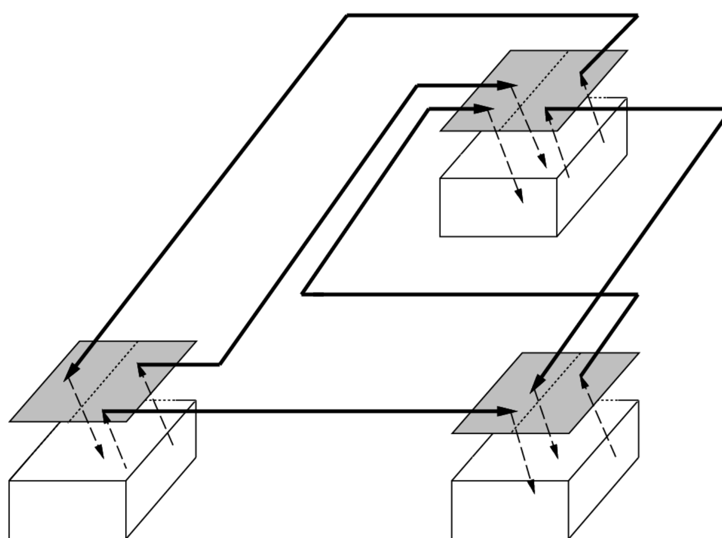
Som følge av at en modul må være helt ferdig designet før den kan beskrives i et ADL gjør det at denne formen for modul beskrivelser er lite aktuelt for AADL. Dette gjør oppsett av en arkitektur i AADL vanskelig på et tidlig tidspunkt i en utviklingsprosess og kan i

så måte være et hinder.

Figur 6.4 på side 29 viser et OCA grensesnitt. I dette tilfellet eksisterer det et grensesnitt, men OCA har kun en modul pr grensesnitt.

6.4.2 ICA

ICA vil si at det er kun grensesnittet som blir beskrevet og gitt tilgang til. Dette gjøres ved å kun si noe om hva en modul tilbyr til sine omgivelser og hva den krever av de samme omgivelsene.[21] Figur 6.5 på side 30 viser en grafisk oversikt over hvordan ICA er ment å fungere. Her er grensesnittet løsrevet fra selve funksjonaliteten i modulen, og videre er det delt opp i hva modulen tilbyr og hva den krever. Fordelen med at grensesnittet er løsrevet fra selve funksjonaliteten er at ved en oppdatering vil man kunne bruke samme AADL fil siden det ikke vil finnes referanser ned til oppførsel, men kun til funksjonalitet. Med et godt definert grensesnitt kan selve funksjonaliteten under endres uten at grensesnittet endres. Dette vil si at to moduler uten annen sammenligning enn felles grensesnitt vil kunne benytte samme grensesnitt.



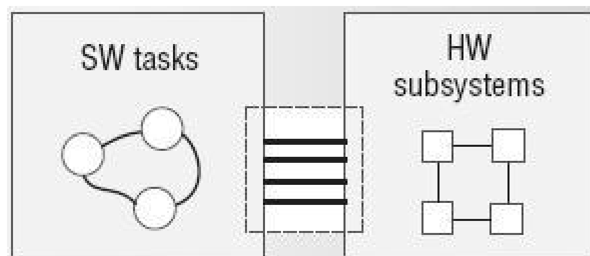
Figur 6.5: ICA[2]

Formålet ved å benytte ICA i AADL er å kutte koden ned til et minimum for en designer å kode og gjøre det enklest mulig å tolke

for AHEAD. To moduler vil ikke kunne benytte seg av samme AADL beskrivelese selv om ICA er valgt som grensesnittbeskrivelse siden en modul i AHEAD vil ha endringer i areal, prosesseringskapasitet, tilbud og krav.

6.4.3 Sw grensesnitt mot HW

I kapitell 6.2.9 nevnes noen generelle krav SW har til den HW'en som skal eksekvere SW. Sw setter også krav til det grensesnitt som eksiterer mellom HW og SW. I AHEAD er det kun valgt en plattform, tag, som er en FPGA og denne representerer all HW. Grensesnittet mellom HW og SW kan varierer fra modul til modul. En standarisert måte å beskrive dette grensesnitte på i AADL blir derfor problematisk med tanke på et høyt abstaksjonsnivå.



Figur 6.6: Sw HW standarisert grensesnitt.[4]

Figur 6.6 viser et standarisert grensesnitt mellom SW og HW. Selve overgangen, som sett i figur 6.6, kan være standarisert arkitektur i HW modulen eller et standard operativsystem. Med standarisert arkitektur menes den delen som er synlig for SW og dette er først og fremst instruksjonssettet. Instruksjonssettet fører til et abstrakt grensesnitt hvor data flyten i HW blir usynlig for SW'en.[4]

Instruksjonssettet til en prosessor forteller ikke så mye om en prosessorsegenskaper som arkitekturen. Så i AADL vil SW være knytte opp til arkitektur og ikke instruksjonssett ved valg av HW.

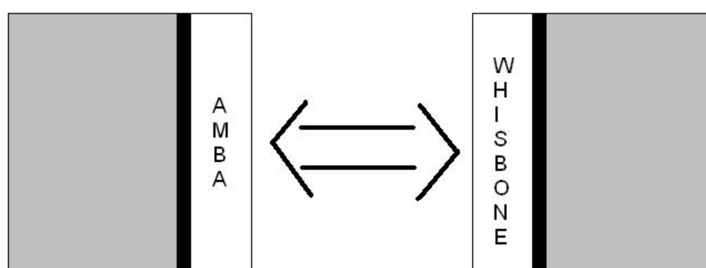
Navnerom

Sw har derfor i lik grad med HW behov for et navnerom hvor instruksjonssett, HW-arkitektur og/eller operativsystem er predefinert ved

hjelp av et navnerom. Dette vil føre til et liknende abstraksjonsnivå for SW-beskrivelse i AADL som HW.

6.5 Kommunikasjonsprotokoller

Kommunikasjonsprotokollen i AHEAD prosjektet ser i skrivende stund ut til å bli pakkeruting gjennom mux'er. AADL vil ikke beskrive kommunikasjonsmodellen i de enkelte arkitekturer/systemer i minste detalj. Dette vil si at kommunikasjonsprotokollene i hovedsak må være predefinerte for at AADL skal klare å behandle modulene. Gjennom et ICA grensesnitt for AADL vil predefinerte protokoller være enkelt å formidle både til maskin og designere. Selv ved en nøyaktig beskrivelse av en kommunikasjonsprotokoll i AADL vil ikke en maskin være i stand til å bruke denne informasjonen for å tilpasse en annen modul til å kommunisere på samme vis. Predefinerte protokoller gjør det lettere for en designer å gjenbruke modulen da man slipper å forholde seg til skreddersydde kommunikasjonsgrensesnitt for hver enkelt modul. I tillegg vil en predefinert protokoll være seg pakkeruting eller en annen protokoll være enkelt å forholde seg til ved design av nye moduler.



Figur 6.7: Grensesnitt eksempel

Figur 6.7 viser et eksempel på to moduler som skal kobles sammen hvor kommunikasjonsprotokollene ikke er i samsvar. Her kan det svarte området på hver modul sees som et standard grensesnitt mellom bussprotokoll og et ferdig definert grensesnitt for modulen. Den svarte området er da et tillegg designerne må forholde seg til men har den fordelen at et program slik som AADL automatisk kan

bytte ut grensesnittet. Ulempen med denne teknikken er at modulen nødvendigvis blir mer komplisert, krever større plass og at designeren uansett må forholde seg til et predefinert grensesnitt.

En predefinert protokoll kan føre til samme problem med navn som ligger i kapitell 6.2.1 på side 21 ha mulighet for å oppstå. Dette vil kunne løses på samme måte med et navnerom.

6.5.1 SW synkron - asynkron kommunikasjon

SW kommunikasjon kan opptre både som asynkron eller synkron kommunikasjon uavhengig om HW som blir brukt. Med synkron SW kommunikasjon menes at det følger en anmodning om kommunikasjon fulgt av en bekreftelse fra mottaker. I en asynkron SW kommunikasjon vil ikke sender bry seg om å be om en bekreftelse fra mottaker men sende data direkte uten å sjekke om mottaker er klar til å motta eller om data'en kommer frem. AADL vil ikke si noe SW kommunikasjonen, men la predefinerte kommunikasjonsprotokoller avgjøre dataflyten innad i arkitekturen.

Kapittel 7

Kodeforklaring og AADL eksempler

7.1 AADL kode eksempler

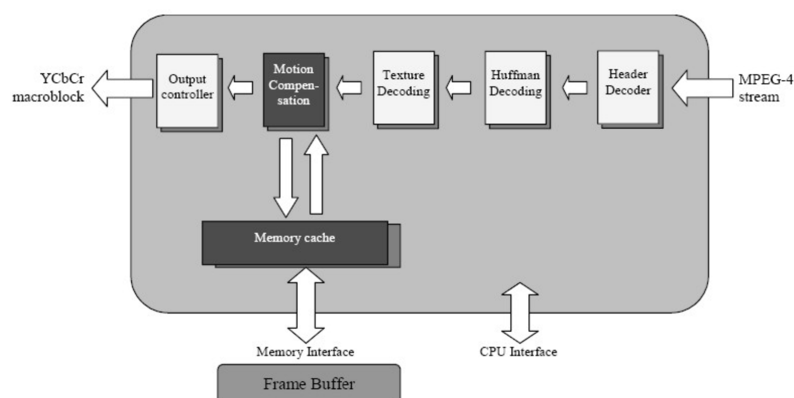
Selve AADL koden vil bli kommentert, men strukteren på hvordan koden er bygd opp vil ikke forklares nøye da denne trolig uansett må endres på ved en implementasjon av AADL. Strukturen inneholder i stor grad ikke informasjon som er nødvendig for AHEAD. AADL er bygget opp på en topp-ned strategi og inneholder blokker hvor funksjonen over blokken er den styrende. Topp-ned er valgt med hensyn på at de er enklere å lage for hånd[15].

7.1.1 Mpeg-4 dekoder

En beskrivelse av MPEG-4 Simple Profile Decoder fra Barco-Silex.[5] Koden er ikke beregnet for AHEAD prosjektet da det ikke finnes moduler for dette prosjektet enda og blir av denne grunn beskrevet ut fra databladet. Dette vil gi mangelfull AADL kode, som ikke vil kunne brukes direkte til en eventuell kompilering av modulen inn i et AHEAD system.

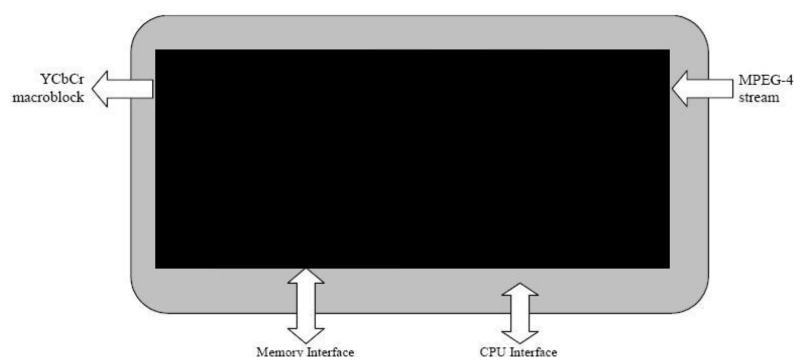
Figur 7.1 viser hvordan en MPEG4 dekoder fungerer. Her ser man et grensesnitt og en enkel oppførselsbeskrivelse som viser datagangen innad i dekoderen.

Databladet gir noe mer informasjon om virkemåte og hvordan de forskjellige grensesnittene er strukturert rent HW messig. Infor-



Figur 7.1: Barco Silex MPEG4 Dekoder.[5]

masjon om datasammensetning for de forskjellige grensesnittene er ikke oppgitt. [5]



Figur 7.2: Barco Silex MPEG4 Dekoder, svart boks.[5]

Figur 7.2 på side 36 viser hvordan AADL er ment å beskrive en modul. Her er selve innholdet fjernet med en svart boks mens grensesnittet står igjen og er løsrevet fra selve oppførselen. Oppførselen blir kun beskrevet som tilbud og krav til systemet som en helthet i de tilfeller hvor dette ikke kommer klart frem gjennom grensesnittsbeskrivelsen.

Barco Silexs MPEG4 dekoder AADL-kode

```
// Generell data om krets.
MODULE(HW);
```

```
begin(
TYPE(CUSTOM);
FILE(BSHWMPEG4DEC)            // Filnavnet på bitfilen.
AREAL(5450, -1, -1, -1, -1); // tall beregnet for Xilinx
                             // XC2V1500-4
CLOCK(1,65);
INTERFACE(4,(STREAM,IN,CUSTOM,1,1),(STREAM,OUT,CUSTOM,2,1),
            (CUSTOM,INOUT,CUSTOM,3,1),(CUSTOM,INOUT,CUSTOM,4,1));


// Absolutte krav til systemet.
REQUIRES();
begin(
    MPEG4
)end;

// Tilbud til systemet
PROVIDES();
begin(
    YCbCr
)end;

// Hardware beskrivelser, sammenkopling med resten.
PORT();
begin(
        (CLK, IN, 1, 3),
        (RESET, IN, 1, 3),
        (START, IN, 1, 3),
        (DONE, OUT, 1, 3),
        (ERROR, OUT, 1, 3),
        (ERRORCODE, OUT, 16, 3),
        (CSTRB, IN, 1, 1),
        (CSTUM, IN, 3, 1),
        (CD, IN, 8, 1),
        (CFULL, OUT, 1, 1),
```

```
        (PFULL, IN, 1, 2),
        (PD, OUT, 8, 2),
        (PSTRB, OUT, 1, 2),
        (MRQFULL, IN, 1, 4),
        (MRQPUSH, OUT, 1, 4),
        (MRQADDR, OUT, 32, 4),
        (MRQEMPTY, IN, 1, 4),
        (MRQPOP, OUT, 1, 4),
        (MRQRD, IN, 32, 4),
        (MWQFULL, IN, 1, 4),
        (MWQPUSH, OUT, 1, 4),
        (MWQADWD, OUT, 32, 4),
)end; // Avslutter PORT

)end; // avsluttning til MODULE
```

Kommentarer

I dette tilfellet vil MPEG4 dekoderen som vist overfor ha et krav til systemet og det er at MPEG4 blir gitt til systemet. YCbCr er et tilbud gitt fra systemet, og dette er de ferdige dekodede data og kan sendes ut til den håndholdte enheten.

7.1.2 Sw eksempel

Sw eksemplet er hentet på www.avrfreaks.net og er kodet for en AVR-kontroller. Denne modulen krever tilgang til 7 LED lamper og SW'en vil kontrollere disse. Et lite eksempel som dette vil vise de mest grunnleggende behovene for SW i AADL.

Simpledice fra avrfreaks

```
MODULE(SW);
begin(
FILE(dice.hex);
MEMORY(1);     // Minneforbruk.
```

```
// tilbud og krav til hw fra sw'en
PROVIDES();
begin(
    6D,     // Gir en 6 sided terning simulasjon.
)end ;

REQUIRES();
begin(
    AT90S1200,   // mikrokontroller arkitektur.
    7-LED,      // Krever 7 LED i HW.
)end;

)end; // avslutter module
```

Kommentar

En SW modul er alltid koded for en spesiell arkitektur og lage en generell tilpassning for dette i AHEAD vil være vanskelig. De fleste mikrokontroller har særegenheter ved sin arkitektur som gjør at kode skrevet spesifikt for en prosessorarkitektur ikke kan brukes på andre arkitekturer eller blir vanskelig å konvertere automatisk.

7.2 Kodeforklaring

7.2.1 Kodeoppdeling

Koden er tredelt. Hver enkelt del representerer sin del av en HW eller SW modul.

1. Genrelle krav til systemet som helhet
2. Krav og tilbud fra modulen
3. HW oppkobling

Del 1 gir de krav som systemet må klare å gi, som riktig klokkefrekvens og stort nok areal. Disse kravene må bli fulgt hvis modulene skal med 100% sikkerhet fungere som tenkt.

Del 2 gir de krav som systemet som ligger på AHEAD tagen for at modulene skal kunne ha sin opprinnelige funksjonalitet. Disse kravene sikrer ikke at modulen fungerer korrekt, men sikrer at oppførselen kommer til nytte.

Del 3 viser kun hvilke grensesnitt som er koblet opp mot hvilke HW porter. For en SW modul vil har ikke denne delen noe tilsvarende innhold, men kan inneholde for eksempel krav til plassering i minne i forhold til oppstart av programmet.

7.2.2 Plassering i HW

Det blir ikke presentert metoder for å flytte en modul innad på en FPGA i AHEAD ved hjelp av AADL. Dette er gjort fordi det enda ikke er vist hvordan en modul kan flyttes på og hvordan grensesnittet innad på en FPGA vil fungere og derfor blir det vanskelig å lage metoder eller beskrive en slik flytting.

7.2.3 HW og SW i samme AADL

Kapitell 6.2.7 sier at AADL skal være til hjelp ved en automatisk valg av SW og HW der begge eksisterer for en modul.

Eksempel på HW og SW

```
// SW beskrivelse
MODULE(SW);
begin(
// beskrivelsen av modulen.
)end;

// HW beskrivelse
MODULE(HW);
begin(
// beskrivelsen av modulen.
)end;
```

Fordi begge moduler allerede eksisterer som forskjellige kilder holdes beskrivelsene separat, men ved å holde de i samme AADL fil blir det sagt at de har samme funksjonalitet og tilbud.

Sammenligning av tilbud i forskjellige moduler vil kun si om to moduler er like. Ulempen med dette er at tilbudsdelen av AADL koden er skrevet av en designer og kan derfor ikke med sikkerhet si om to moduler gir identisk funksjonalitet. Derfor vil et prinsipp med en AADL fil med to beskrivelser være en sikkerhet for at de fungerer likt.

Ved å beskrive HW modulen og SW modulen i hver sin AADL fil vil kreve at AHEAD vil kunne sammenligne filene og oppdage at de har samme funksjonalitet. Denne operasjonen blir ungått ved å sette de i samme AADL-fil.

7.2.4 Antall grensesnitt

En modul kan ha flere grensesnitt mot systemet. Disse kan operere på flere forskjellige frekvenser og/eller ha forskjellige formater mot systemer. Det er derfor nødvendig å skille disse grensesnittene fra hverandre i AADL for å kunne lettere håndtere disse lettere. Dette er forsøkt gjort med funksjonen Interface(). Et eksempel på en modul med to grensesnitt er en Amba AHB bro som operer på to forskjellige frekvenser.

7.3 AADL begrensninger

De største begrensningene til AADL er behovet for behandling av AADL filer på AHEAD. AHEAD arkitekturen må kunne behandle AADL filer og dette vil oppta ressurser på FPGA'en.

En modul vil kunne gjenbrukes i andre systemer så lenge alle tilbud og krav, grensesnitt og kommunikasjonsprotokoller er møtt. For å øke gjenbruket av moduler må kun et begrenset antall grensesnitt og kommunikasjonsprotokoller benyttes.

Kapittel 8

Konklusjon

AADL bør være et høy abstraksjonsspråk som kun skal brukes til en automatisk plassering av moduler på en FPGA. Grensesnittet til AADL bør benytte ICA fordi dette er et grensesnitt som er lett å behandle opp mot plassering innad på en FPGA, og ta hensyn til i et genrelt system. Oppbyggningen av en AADL-fil bør gjøres etter et sjekklisteprinsipp, og en dobbelkode ala Javadoc i den funksjonalitetsbeskrivende kilden bør lages. Dobbelkode vil føre til at en AADL beskrivelse ikke blir statisk ved endring av kilden.

Navnerom på oppførsel, tilbud og krav, og predefinerte kommunikasjonsprotokoller vil være med på å holde AADL på få innebygde funksjoner og fører til en lav læringskurve for designeren. Blokkmiljøet bør implementeres for å holde orden på navnerommet.

Behandling av AADL i AHEAD-arkitekturen er ikke til å unngå, og et høyt abstraksjonsnivå vil kunne bidra til å holde ressursene dette krever til et minimum.

8.1 Fremtidig arbeid

Fremtidig utvikling av AADL bør ikke gjøres før AHEAD prosjektet har en sikker valgt plattform, og en metode for plassering av moduler innad på tagen er utviklet. Gjenbruk av moduler innad i AHEAD ved hjelp av AADL vil være avhengig av en metode for automatisk plassering blir funnet.

Moduler og systemer som inkulderer SW bør være utviklet før

AADL blir definert, for å dra nytte av de erfaringer slike prosjekter drar med seg.

Bibliografi

[1] Stig Petersen, Dag K. Rognlien, and Kjetil Svarstad. *D3 – Context Tag Technology*. SINTEF.

[2] Rapide Design Team. *Guide to the Rapide 1.0 Language Reference Manuals*. Computer System Labs Stanford University, draft edition, Juli 1997.

[3] Xilinx. *Development System Reference Guide*, 2005.

[4] Ahemd A. Jerraya and Wayne Wolf. Hardware/software interface codesign for embedded systems. *Computer*, 38(2):63 – 69, Februar 2005.

[5] Barco-Silex. *Mpeg-4 Simple Profile Decoder BA132MPEG4D Factsheet*. Barco-Silex, Rue du Bosquet 7 B-1348 Louvain-La-Neuve, Oktober 2005.

[6] Christian Feichtner, Erwin Postmann, Hans I. Myrhaug, Ayse Goker, Tor Erlend Fægri, and Børge Haugset. *D1 – Requirements and Overall system architecture report*. SINTEF, 2.0 internal edition.

[7] Katherine Compton and Scott Hauk. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171 – 210, June 2002.

[8] Ingar Hauge. Arkitekturbeskrvelser for ahead. Desember 2005.

[9] Mary Shaw. Truth vs knowledge: The difference between what a component does and what we know it does. Carnegie Mellon

University, 8th International Workshop on Software Specificatin and Design, Mars 1996.

[10] Douglas W. Bennet. The promise of reuse. *Object Magazine*, 4(5):32 – 40, Januar 1995.

[11] Paul C. Clements. A survey of architecture description languages. Carnegie Mellon Univeristy, 1996.

[12] Mary Shaw. When is 'good' enough?: Evaluating and selecting software metrics. *Software Metrics: An Analysis and Evaluation*, pages 251 – 262, 1981.

[13] Martin Grant. Design methodologies for system level ip. *Design, Automation and Test in Europe*, pages 286 – 289, Februar 1998.

[14] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using rapide. *IEEE Transactions On Software Engineering*, 21(4):336 – 355, April 1995.

[15] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers, principles, techniques and tools*. Addison-Wesley Publishing Company, 1986.

[16] Y. N. Srikant and P. Shankar. *The Compiler Design Handbook*. CRC Press, 2000.

[17] Darius Blasband. Parsing in a hostile world. pages 291 – 300. RainCode Corp., Oktober 2001.

[18] Daniel Mesquita, Fernando Moraes, Josè Palma, Leandro Möller, and Ney Calazans. Remote and partial reconfiguration of fpgas: tools and trends. *Computer*, 2003.

[19] Michael Keating. *Reuse methodology manual for system-on-chip design*. Kluwer Academic Publishers, third edition, 2002.

[20] John L. Hennessy and David A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, 3 edition, 2003.

[21] T. Cook. Architecture descriptions languages: An overview, Juli 1999.

Tillegg A

AADL kode forklaring

Oppbyggningen på en funksjon er satt opp som parameter og variable. En parameter skal fylles inn med en predefinert verdi, mens en variabel inneholder den data som beskrives for den enkelte funksjon, et eksempel på dette er MODULE som inneholder en parameter og FILE som inneholder en variabel. En variabel vil ha et variabelnavn i funksjonsbeskrivelsen.

Logikk

En funksjon krever i noen tilfeller fra en til flere parametre eller variable for å utfylle sin funksjon.

|| Logisk eller.

&& Logisk og.

Paranteser behandles først i fra innerste til ytterste parantes.

Variable skal skilles fra hverandre i funksjonen ved bruk av komma. Et eksempel på dette er *CLOCK(4,(25,50,65,53.5));*.

N i en funksjonsforklaring er en del av et funksjon som skal gjentages N ganger. N er forklart i funksjonen. Et eksempel på bruk av N er CLOCK funksjonen.

Blokk

Dette beskrives av en tom funksjon som avgjør blokk type etter fulgt av en **begin(** etterfulgt av variable eller parametre avgjort av funk-

sjon og avslutter blokken med en **)end**.

Blokkfunkjsoner er merket med begin()end i funksjonsinformasjonen.

AADL Funksjonsforklaringer

Påfølgende sider er forklaringer av funksjoner i AADL.

MODULE(Parameter) begin()end

Module definerer om den modulen som blir beskrevet eksisterer i SW og/eller HW. Alt informasjon om en spesifik modul skal ligge i scopet til denne funksjonen.

Parametere

SW Brukes når AADL har vedlagt eksekverbar SWkode.

HW Forteller designeren at dette er en hardware beskrivelse.

FILE(Navn)

File inneholder navnet til den filen som beskrives.

Variabel

Navn Navnet på filen som beskrives.

TYPE(Parameter)

Type beskriver hva slags funksjonalitet denne modulen har. Det vil eksistere tilfeller hvor type avgjør plassering av hardware på FPGA'en eller prioritering av software i minnet.

Parametere

FILTER Modulen er et filter.

PROCESSOR Modulen er en prosessor.

MEMORY Minnemodul.

CUSTOM Modulen innehar en ikke definert funksjonalitet.

AREAL(SLICE && CLB && LUT && FF && RAM)

Beskriver arealet som trengs på FPGA'en. Dette kan enten beskrives ved hjelp av FPGA'en innebygde logikk eller ved hjelp av selve arealet i mm^2. Sett alle ukjente verdier til -1.

Variabel

SLICE Antall SLICEs modulen krever.

CLB Antall CLBs modulen krever.

LUT Antall LUTs modulen krever.

FFs Antall FFs modulen krever.

RAM Antall block RAMs som modulen bruker.

REQUIRES() begin(Krav)end

Funksjonen setter et krav til at designeren har laget eller sett på et navnerom, og funnet eller lagt til sine krav.

Variabel

Krav Navnet til kravet som modulen har.

PROVIDES() begin(Tilbud)end

Funksjonen setter et krav til at designeren har laget eller sett på et navnerom, og funnet eller lagt til sine tilnud.

Variabel

Tilbud Navnet til tilbudet som modulen har.

MEMORY(Minne)

Brukes til å si hvor mye minne en SW modul krever.

Variable

Minne Antall KB som modulen bruker.

CLOCK(#Klokker && (N Frekvenser))

Clock oppgir antall klokker modulen har behov for og hvilke frekvenser disse skal ha. Andre halvdel skal inneholde like mange frekvenser som det finnes klokker.

Variabler

#Klokker Antall klokker.

N Frekvenser Her skrives frekvensene til de forskjellige klokkene, oppgis i MHz.

Eksempel på bruk

Fire klokker på 25, 50, 65, 53.5 MHz skal oppgis for en modul. Her er N 4.

CLOCK(4,(25,50,65,53.5));

INTERFACE(#interface && N(Parameter1 && Parameter2 && Standard && #nr && #klokkkenr))

Denne beskriver hvor mange grensesnitt en modul innehar. Den er mest rettet mot HW moduler.

N forteller hvor mange ganger parentesen skal repeteres. N er det samme som #interface.

Variable

#interface Antall grensesnitt modulen innehar.

Standard Dette oppgir busstandarden til dette grensesnittet. Hvis ukjent oppgi CUSTOM.

#nr Hvilket nummer dette grensesnittet har utover i modulen. Viktig i forhold til hvor porter skal kobles. Rekkevidden er 1 til #interface.

#klokkenr Hvilken klokkefrekvens grensesnittet skal ha.

Parameter1

Beskriver formatet

STREAM Data kommer eller blir levert som en strøm. Krever at kommandoen STREAM blir etterfulgt med informasjon om data strømmen.

BUS Data blir levert over et standard bussgrensesnitt.

CUSTOM Spesialtilpasset grensesnitt.

NONE Denne modulen har ikke behov for kontrolldata.

Parameter2

IN Modulen har kun data inn gjennom dette grensesnittet.

OUT Modulen har kun data ut gjennom dette grensesnittet.

INOUT Toveis grensesnitt, kan motta og sende data.

PORT() begin(Navn, Parameter1, #bredde, #interfacenr)end

Beskriver hvilke porter som er tilkoblet hvilket grensesnitt og hvor de skal kobles.

Parameter1

Beskriver retning på porten.

IN Dataflyt inn.

OUT Dataflyt ut.

INOUT Data kan gå både inn og ut.

Z Tristate port.

Variable

Navn Navn på porten i grensesnittet.

#bredde Antall bit på porten.

#interfacenr Hvilket nr grensesnitte har i interface funksjonen.