Armor -
Langage de description d’architecture

États des lieux, perspectives
Séminaire COSI
Roscoff – juin 2001

ADL
Architecture Description Language
Why Architecture Description Language?

- SOC technology enables integration of processors, memory into one chip
- More and more tasks are shifted into programmable parts
- Short time to market requires the use of higher level language for application description
- Customisation of processor-memory system produces more efficiency, design space exploration is needed
- Software toolkit development for ASIP is time consuming, automatic generation is desired
- ADL is a bridge between architecture exploration and software toolkit generation

Architecture Description Language Overview

- Desirable features
  - Simple and natural
  - General : wide range of architecture support
  - Support automatic toolkit generation
  - Compiler, simulator
  - Synthesize
  - Not much human interventions

- Classification
  - Structural level : components, connection
  - Behavioural level : instruction set
  - Mixed level : combination of both
Structural level ADLs

- **MIMOLA**
  
  Used by MSSQ and Record compiler
  
  RTL level: a single description can be used for
  synthesis, simulation, test generation and code
  generation
  
  No explicit support for pipelines or resource conflicts

- **XASM**

  Used in BUILDABONG project
  
  Abstract state machine generated automatically from a
  graphical description
  
  Only for simulator, instruction set information not
  included

---

Behavioural level ADLs

- **nML**

  Used by code generator, instruction set simulator
  
  assembler/disassembler in IMEC, Cadence
  
  Mainly describes instruction set
  
  Weak structural description

- **ISDL**

  Used by SPAM project
  
  Explicit constraint between operations
  
  Weak timing model
  
  No structural information except storage
Mixed level ADLs

- Flexware
  Instruction set, resources, interconnects
- LISA
  Developed at Aachen, being used in industry
  Assembler, linker, simulator, (compiler)
  Support synthesis
  Detailed pipelined transactions
- Expression
  Explicit instruction set behaviour and code mapping specification
  Netlists of components, can extract reservation table
  target VLIW, DSP, superscalar, …

Mixed level ADLs (cont’d)

- PRML
  Limited scope
  Used in Philips, for CPU64 core in Trimedia
- HMDES
  General purpose processor oriented
  Reservation table based
  Instruction set information is incomplete
Current status of ADLs

- No standards
- Serves directly, and in most cases only, for the compiler and simulator in its own community
- Rough description easy, complete description hard
  - Tradeoff between language simplicity and the huge target space: ASIP, DSP, VLIW, superscalar
  - Unexpected new architecture features/constraints
  - Irregular constraints
  - Exception
  - Conditional execution specified?

To think about

- Architecture range
  - Support for compiler, simulator
- Instruction set architecture constraints
  - ILP Constraint
  - Operand constraints
  - Mixed Constraints (inter-operation constraints)
- Reservation table versus netlists
  - Conciseness
  - Intuitiveness
  - Easiness to modify
ARMOR Architecture Description
Language

Armor language

- Behaviour of the instruction-set
  - operational semantics
  - resource usage
  - internal parallelism
  - No information on forms of instructions

- Short description
  - inspired by the nML principles
Instruction-set structure

Tms320c6x

instructionSet =
[ inst || inst || inst || inst || inst
  || inst || inst || inst ]

gp inst = add | sub | ...

Data flow

df asmul is { r=mul(adsub(r,r),r) }
op adsub = add | sous
op add(x,y) is ADD(x,y) <ress=ALU>
op sous(x,y) is SUB(x,y) <ress=ALU>
op mul(x,y) is MUL(x,y) <ress=MP>

fu ALU <cycle=1> fu MP <cycle=2>
regFile r(16,word) <access Rd=1 Wr=2>
type word is integer(32)
Address generation unit

\[
\text{df load is } \{r=\text{memory}[ad]\}
\]

\[
\text{address ad = simple} \\
| \text{postmod}
\]

\[
\text{address simplep is AR} \\
\text{address postmod is AR}\%i\{\text{opAd(AR}\%i, \text{src})\}
\]

\[
\text{op opAd = inc} \mid \text{dec} \\
\text{op inc(x,y) is ADD(x,y) } \langle \text{res} = \text{AGU}\_\text{ALU}\rangle
\]

\[
\text{mode src = IR} \mid \text{const(1)}
\]
// register definition
regFile X(32 * 32) <access Rd=1 Wr=1> <nbAccess Rd=2 Wr=2>
regFile Y(32 * 32) <access Rd=1 Wr=1> <nbAccess Rd=2 Wr=2>
regFile Rs(sword) alias[X]
regFile Ru(uword) alias[X]
regFile Rt alias[Rs]
regFile RsLoad(sword) alias[X] < access Wr = 1.2>
regFile RuLoad(uword) alias[X] < access Wr = 1.2>
regFile AR(memoryAddress) alias[X]
reg SP(32) <access Rd=1 Wr=1>
execStack ES(M,SP)

// flags
flag fzero
flag fneg

// functional units
fu alu <cycle=1> <set fzero = RESULT_ZERO
  fneg = RESULT_NEGATIVE>
fu mp <cycle=1>

// instruction set
instructionSet = [computation || transfer || transfer ] |
  [ transfer || transfer ] |
  [computation || transfer] |
  computation | comparison | transfer |
  bincond | bcond |
  adModeExtra |
  loop

dataflow component
gp computation = compSign | compUnsign | compMix | addmul
df compSign is { Rs = operation(Rs,source2_sig) }
df compUnsign is { Ru = operation(Ru,source2_unsig) }
df compMix is { Ru = operation(Ru,source2_sig) }
df addmul is { Rs = sum(Rs,multiplication(Rs,Rs)) }
df comparison is { Rs = compare(Rs,Rs) }
mode source2\_sig = Rt \mid \text{imm(int5)} \mid \text{const(1)}
mode source2\_unsig = Ru \mid \text{imm(unsig5)}

gp transfer = moveConst \mid \text{load} \mid \text{store}

gp adModeExtra = moveSymbad \mid \text{loadPointer} \mid \text{storePointer} \mid \text{addArray}

gp moveConst = moveConst1 \mid moveConst2 \mid moveConst3
df moveConst1 is \{ Ru = \text{imm(unsig5)} \}
df moveConst2 is \{ Ru = \text{imm(unsig5)} \}
df moveConst3 is \{ Ru = \text{imm(uword)} \}

gp load = load1 \mid load2
gp store = store1 \mid store2

df load1 is \{ RsLoad = Ms[myAd] \}
df store1 is \{ Ms[myAd] = Rs \}
df load2 is \{ RuLoad = Mu[myAd] \}
df store2 is \{ Mu[myAd] = Ru \}

\text{df moveSymbad is} \{ \text{AR} = \text{imm(memoryAddress)} \}
\text{df loadPointer is} \{ \text{AR} = \text{Mpointer[myAd]} \}
\text{df storePointer is} \{ \text{Mpointer[myAd]} = \text{AR} \}

\text{// addressing modes}
\text{address myAd = immediate} \mid \text{registerAd} \mid \text{postinc} \mid \text{stackAd}
\text{address immediate is} \text{imm(unsig5)}
\text{address registerAd is} Ru
\text{address postinc is} Ru \{ \text{sum(Ru,const(1))} \}
\text{address stackAd is} \text{sum(SP,imm(unsig5))}

\text{// control components}
\text{ctr bincond is BRANCH(imm(branchAddress)) <delay=3>}
\text{ctr bcond is IF genCond THEN BRANCH(imm(branchAddress))}
\text{<delay = 3>
// conditions
cond genCond = b1 | b2 | b3 | b4
cond b1 is TRUE(fzero) <ress = alu>
cond b2 is TRUE(fneg) <ress = alu>

cond b3 is FALSE(fzero) <ress = alu>
cond b4 is FALSE(fneg) <ress = alu>

// loop mechanism
gp loop = rep | bkrep
reg REPc(uword) <access Rd=1 Wr=1> // repeat counter
reg LC(uword) <access Rd=1 Wr=1>    // loop counter
reg LA(uword) <access Rd=1 Wr=1>    // loop address
ctr rep is LOOP <LOOP_COUNT REPC = loopc>
ctr bkrep is LOOP <LOOP_COUNT LC = loopc>
  <LOOP_END LA = loopsize>
mode loopc = Ru | imm(uword)
mode loopsize = imm(uword)

// operators
op operation = sum | difference | multiplication | mod
op compare = sup | inf | supeg | infeg | egal | diff
op sum(x,y) is ADD(x,y) <ress=alu>
op difference(x,y) is SUB(x,y) <ress=alu>
op multiplication(x,y) is MPY(x,y) <ress=mp>
op sup(x,y) is GRT(x,y) <ress=alu>
op inf(x,y) is LESS(x,y) <ress=alu>
op supeg(x,y) is GRT_EQ(x,y) <ress=alu>
op infeg(x,y) is LESS_EQ(x,y) <ress=alu>
op egal(x,y) is EQ(x,y) <ress=alu>
op diff(x,y) is DIFF(x,y) <ress=alu>
op mod(x,y) is MOD(x,y) <ress=alu>
CALIFE Platform

CALIFE foundations

- Create generic transformations
- Offer a way to parameterise transformations (processor description)
- Offer a way to assemble transformations (unified platform)
CALIFE library

Information loaded at runtime

Algorithms

Tool-box

Program representation

Central Model

Sub-models

Gateways

Armor

CALIFE in summary

Target Processor

Optimised code

Source Code

CALIFE/ARMOR tool

Code selection

Resources allocation

Scheduling
Use of Calife for performance estimation of specialized processor

Séminaire COSI - François Charot - juin 2001

Estimation flow

Application code

profiler

analyser (front-end)

Transformation flow

estimator

results

Processor model

Developed within the CALIFE/ARMOR environment
Experiments

- code selection
- address generation
- scheduling

ARMOR model of OAK core

Code selection

Code:
```c
for (...) {
  y+= h[T-1-i]* x[T-1-i];
  x[T-1-i]= x[T-2-i];
}
```

Processor:
```c
... ...[MUL||ADD] ...
```

1. **MUL** P, Array, Array;
2. **ADD** Acc, Acc, P;
3. **LD** Acc, Array;
4. **ST** Array, Acc;
Address generation

Code:
for (...) {
y+= h[T-1-i] * x[T-1-i];
x[T-1-i] = x[T-2-i];
}

Cycle 1  MUL P,(R1)-1, (R2)-1;
Cycle 2  ADD Acc, Acc, P

Processor:

MUL\|ADD

scheduling

Code:
for (...) {
y+= h[T-1-i] * x[T-1-i];
x[T-1-i] = x[T-2-i];
}

Cycle 1  MUL P,(R1)-1, (R2)-1;
Cycle 2  ADD Acc, Acc, P

Processor:

MUL\|ADD

IRISA
scheduling

Final scheduling :

Cycle 1  \[ \text{MUL } P, (R1)-1, (R2)-1; \text{ ADD } \text{Acc, Acc, P;} \]

DSPStone kernels for OAKDSPCore

[Graph showing performance comparison of different kernels]
Design flow with Armor

Architectural database
- Instruction set
- Reservation tables
- Structural informations

Armor file

analyser

View generator

Status

Library of compilation passes

Armor model

C Code

SUIF front-end
- suifFE
- CodeSel
- Sched
- AdrGen
- AllocReg

optimised assembly code
In summary

- The architecture description is a bridge between the architecture and the software tools
  - Essential role in the design of ASIP where the tools contribute towards
    - Architectural development
    - Architectural deployment

- An evolution of the language towards a mixed behavioural/structural level is desirable
  - Modelling of the control part of the processor
  - Need of descriptions as much complete as possible

- Choice of the compilation framework?