The Sprockell

Jan Kuper

University of Twente
The Netherlands

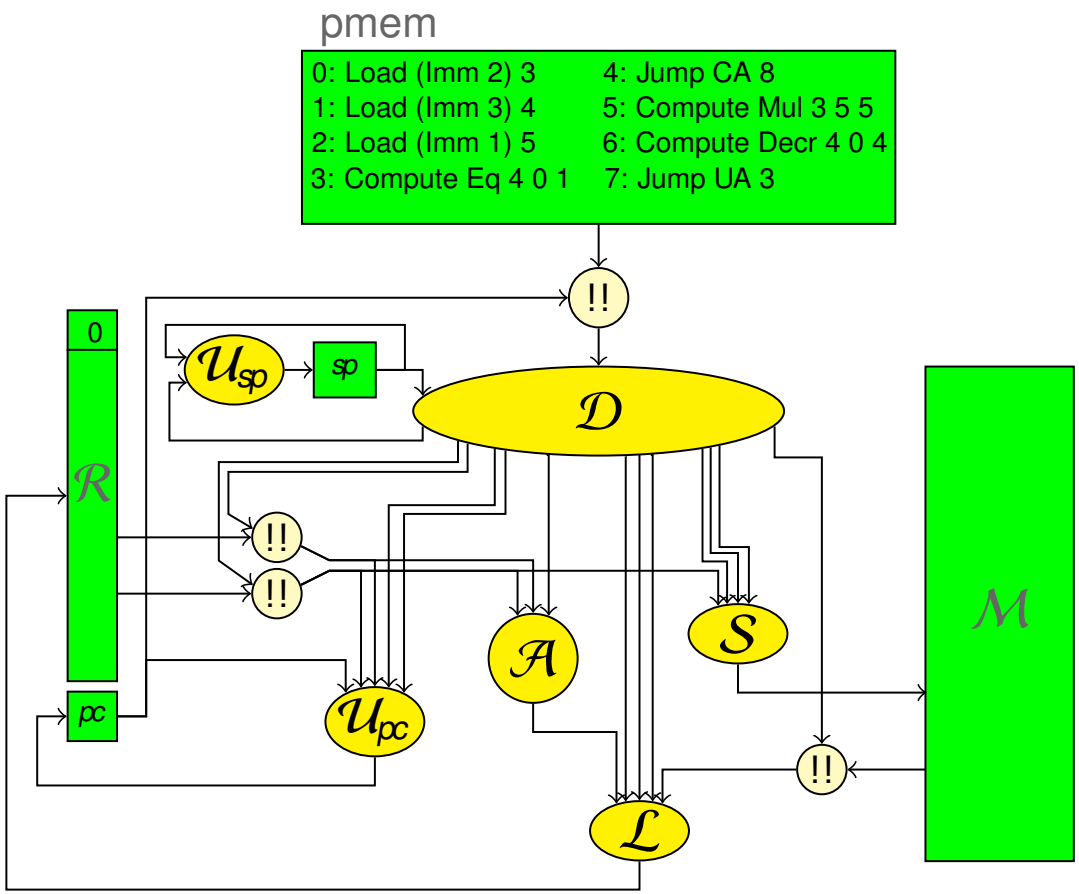
TFPIE 2014 – Soesterberg

May 25, 2014

Introduction

- CλaSH: translating Haskell into VHDL for FPGA/ASIC design (subset, specific hardware types),
- Not only regular architectures (HOFs), also irregular architectures (e.g. processors),
- Experiment: how suitable is Haskell to express such architectures and put that on an FPGA using CλaSH $\Rightarrow$ *Sprockell*,
- Result turned out to be pleasant to use in education, as well for Computer Science students, as for Embedded Systems students — as for lecturers.

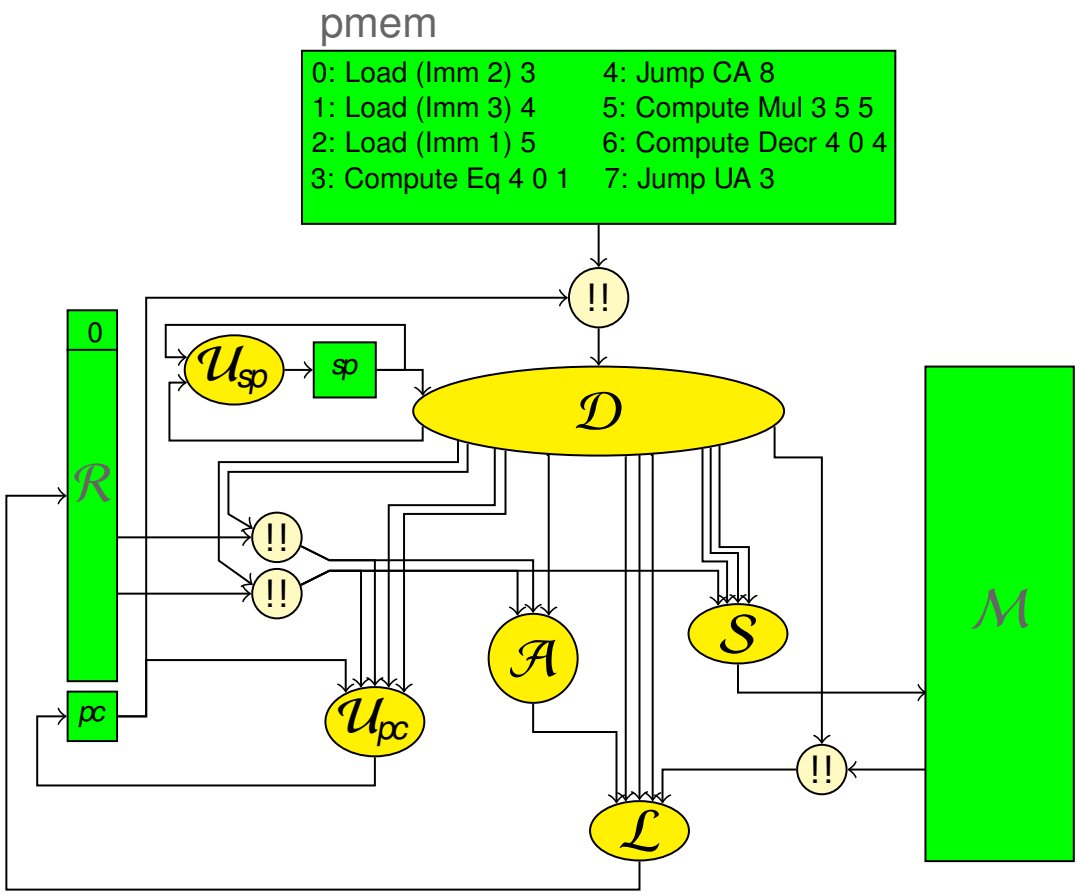
Sprockell: a Simple processor in Haskell



Simplifications:

- one instruction per clock cycle
- no pipelining
- no caches
- no IO
- no $\cdots$

Memory structure



Program memory: [*Assembly*]

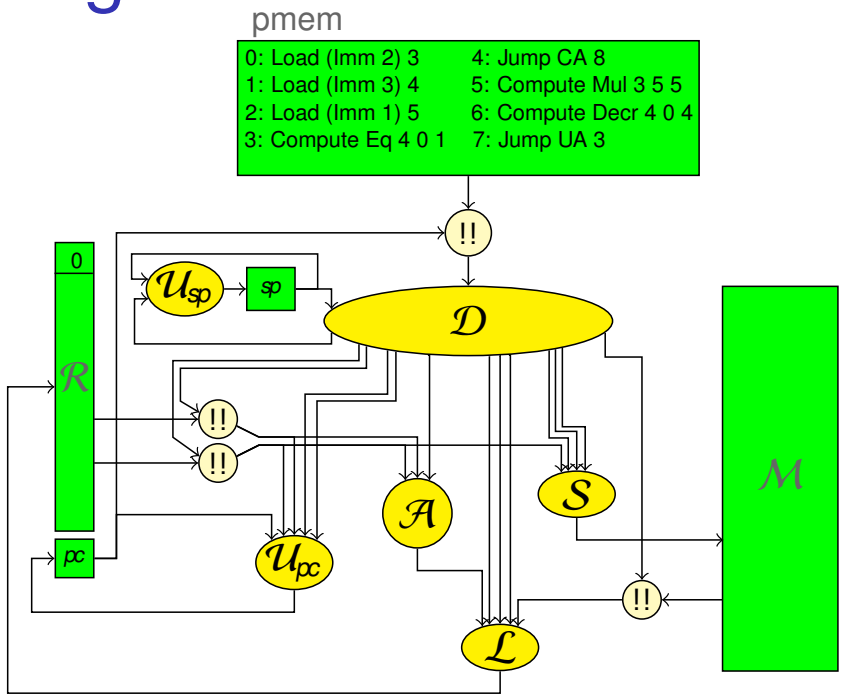
Main memory: $\mathcal{M} :: [Int]$

Register bank: $\mathcal{R} :: [Int]$

Program counter: $pc :: Int$

Stack Pointer: $sp :: Int$

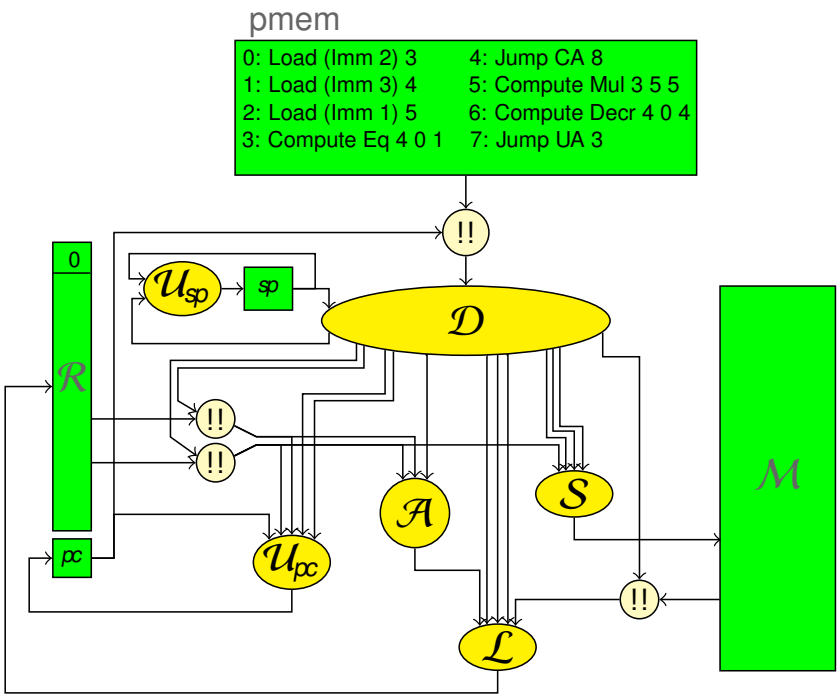
State transforming function



$$\begin{aligned}\textbf{data}\ State = State\ \{&regbank :: [Int]\\ ,&dmem :: [Int]\\ ,&pc :: Int\\ ,&sp :: Int\\ \}&\end{aligned}$$

$$sprockell :: [Assembly] \rightarrow State \rightarrow Clock \rightarrow State$$

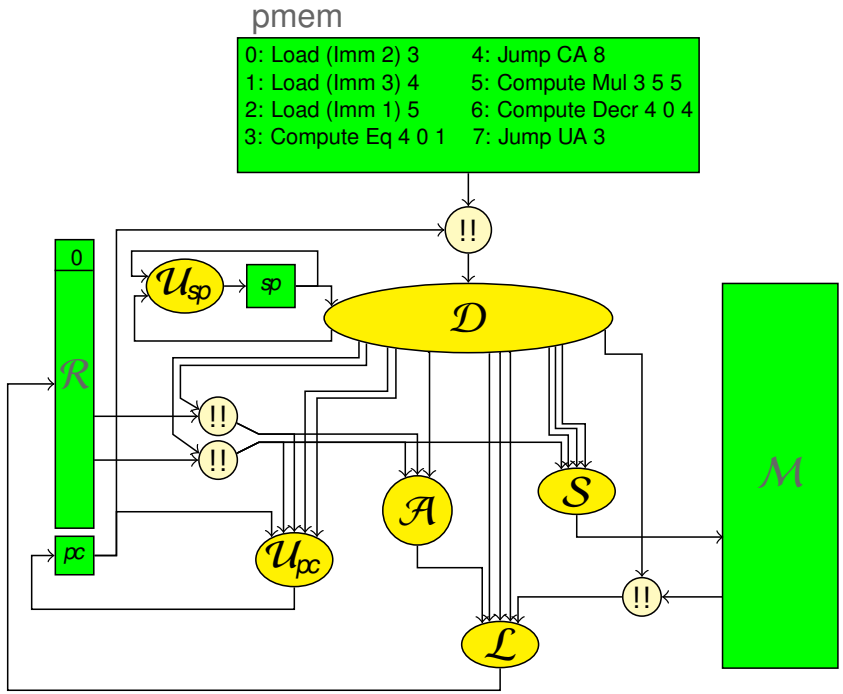
ALU



data *OpCode* = **NoOp** | **Id** | **Incr** | **Add** | **Mul**
| **Equal** | **Gt** | **Not** | **And** | **Or** | ...

$$
\begin{aligned}
&\mathit{alu} :: \mathit{OpCode} \to \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
&\mathit{alu}\ \mathit{opCode}\ x\ y \\
&\quad = \textbf{case}\ \mathit{opCode}\ \textbf{of} \\
&\qquad\qquad \textbf{NoOp} \to 0 \\
&\qquad\qquad \textbf{Id} \to x \\
&\qquad\qquad \textbf{Incr} \to \mathit{incr}\ x \\
&\qquad\qquad \textbf{Add} \to x + y \\
&\qquad\qquad \textbf{Mul} \to x * y \\
&\qquad\qquad \textbf{Equal} \to \mathit{tobit}\ (x \equiv y) \\
&\qquad\qquad \textbf{Gt} \to \mathit{tobit}\ (x > y) \\
&\qquad\qquad \textbf{Not} \to 1 - x \\
&\qquad\qquad \textbf{And} \to x * y \\
&\qquad\qquad \textbf{Or} \to x\ `\mathit{max}`\ y
\end{aligned}
$$

Load



data *LdCode* = **NoLoad** | **LdImm** | **LdAddr** | **LdAlu**

$$load :: [Int] \to LdCode \to Int \to (Int, Int, Int) \to [Int]$$

$$load\ regbank\ ldCode\ toreg\ (immvalueR, mval, z) = regbank'$$

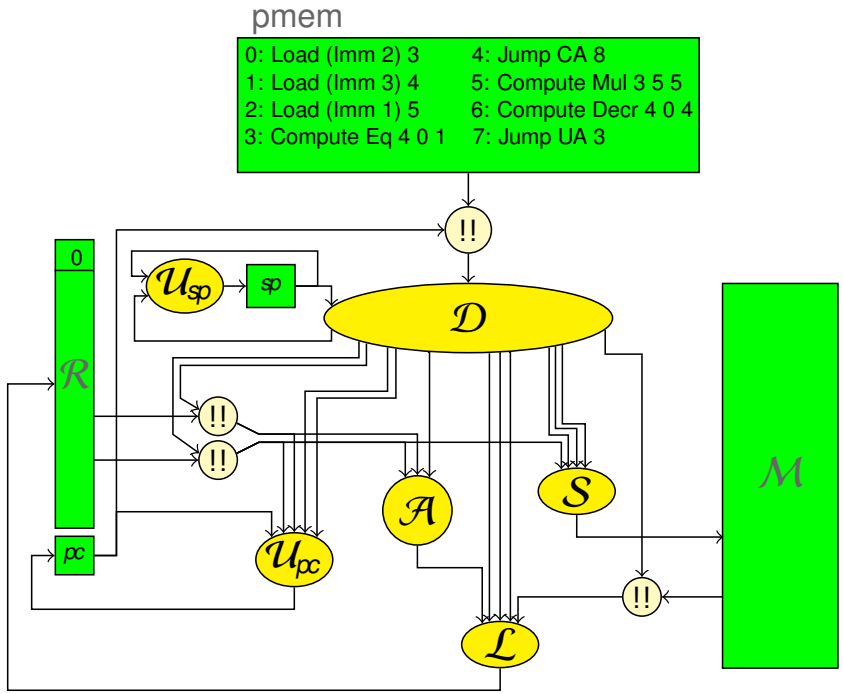
where

$$v = \textbf{case}\ ldCode\ \textbf{of}$$

$$\begin{aligned} &\textbf{NoLoad} \to 0 \\ &\textbf{LdImm} \to immvalueR \\ &\textbf{LdAddr} \to mval \\ &\textbf{LdAlu} \to z \end{aligned}$$

$$regbank' = regbank \leftsquigarrow (toreg, v)$$

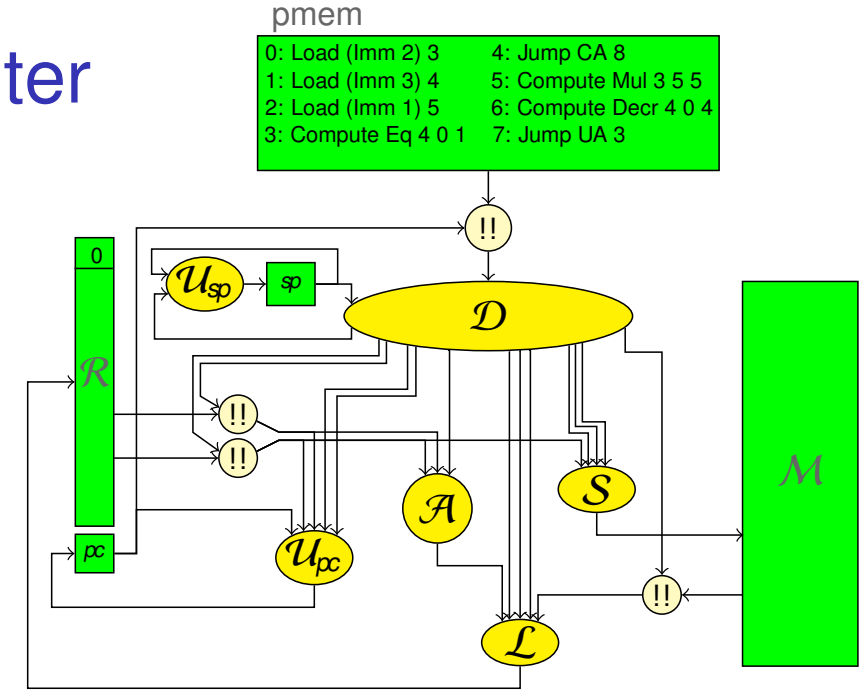
Store



data *StCode* = **NoStore** | **StImm** | **StReg**

$$
\begin{aligned}
&\mathit{store} :: [\mathit{Int}] \to \mathit{StCode} \to \mathit{Int} \to (\mathit{Int}, \mathit{Int}) \to [\mathit{Int}] \\
&\mathit{store}\ \mathit{dmem}\ \mathit{stCode}\ \mathit{toaddr}\ (\mathit{immvalueS}, x) = \mathit{dmem}' \\
&\quad \textbf{where} \\
&\qquad \mathit{dmem}' = \textbf{case}\ \mathit{stCode}\ \textbf{of} \\
&\qquad\qquad\qquad\qquad \textbf{NoStore} \to \mathit{dmem} \\
&\qquad\qquad\qquad\qquad \textbf{StImm} \quad \to \mathit{dmem} \leftsquigarrow (\mathit{toaddr}, \mathit{immvalueS}) \\
&\qquad\qquad\qquad\qquad \textbf{StReg} \quad \to \mathit{dmem} \leftsquigarrow (\mathit{toaddr}, x)
\end{aligned}
$$

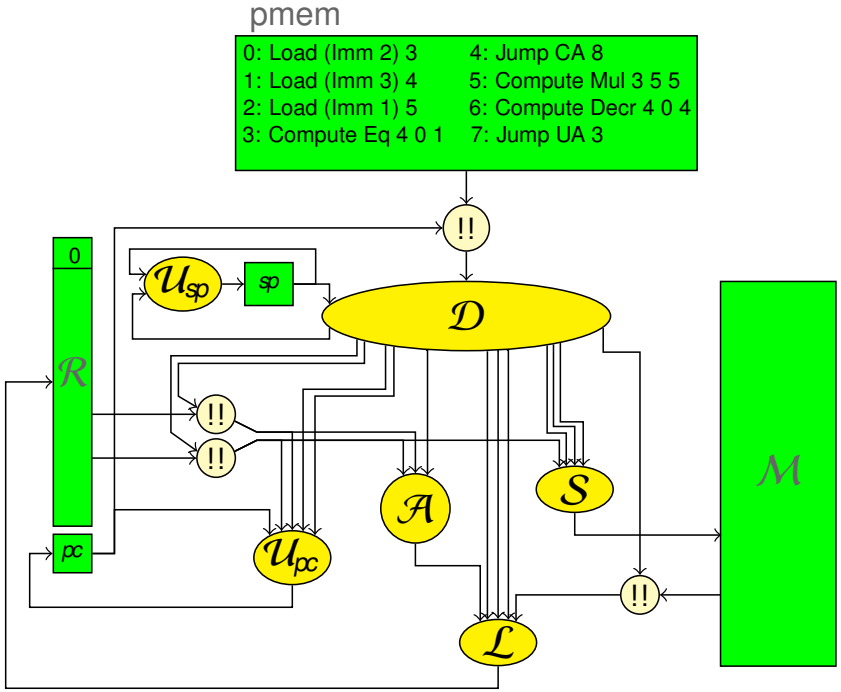
Program counter



data *JmpCode* = **NoJump** | **UA** | **UR** | **CA** | **CR** | **Back**

$$
\begin{aligned}
&\mathit{pcUpd} :: (\mathit{JmpCode}, \mathit{Int}) \to (\mathit{Int}, \mathit{Int}, \mathit{Int}) \to \mathit{Int} \\
&\mathit{pcUpd}\ (\mathit{jmpCode}, x)\ (\mathit{pc}, \mathit{jumpN}, y) \\
&\quad = \textbf{case}\ \mathit{jmpCode}\ \textbf{of} \\
&\qquad\quad \textbf{NoJump} \to \mathit{incr}\ \mathit{pc} \\
&\qquad\quad \textbf{UA} \to \mathit{jumpN} \\
&\qquad\quad \textbf{UR} \to \mathit{pc} + \mathit{jumpN} \\
&\qquad\quad \textbf{CA} \to \textbf{if}\ x \equiv 1\ \textbf{then}\ \mathit{jumpN}\ \textbf{else}\ \mathit{incr}\ \mathit{pc} \\
&\qquad\quad \textbf{CR} \to \textbf{if}\ x \equiv 1\ \textbf{then}\ \mathit{pc} + \mathit{jumpN}\ \textbf{else}\ \mathit{incr}\ \mathit{pc} \\
&\qquad\quad \textbf{Back} \to y
\end{aligned}
$$

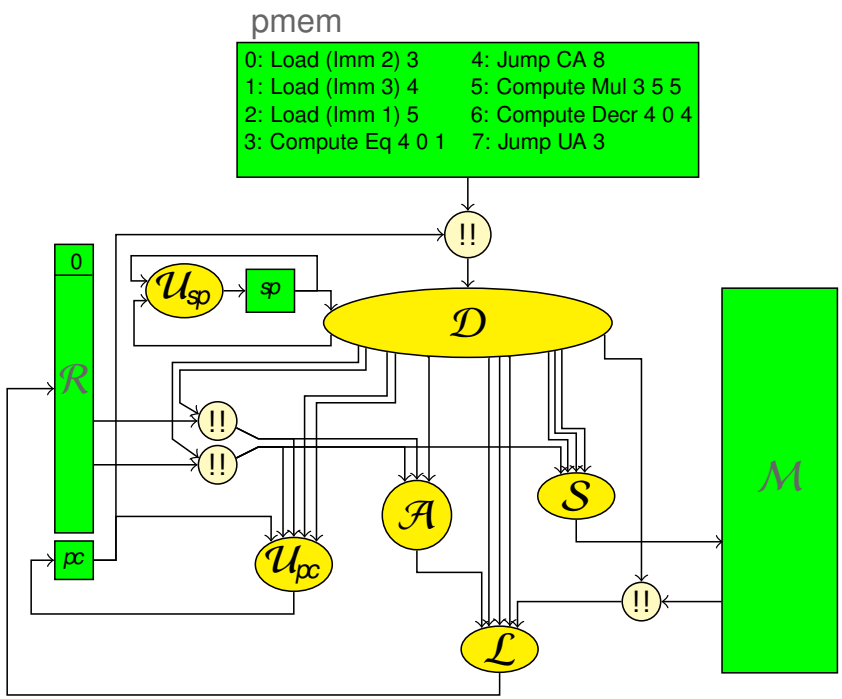
Stack pointer



data *SPCode* = **None** | **Up** | **Down**

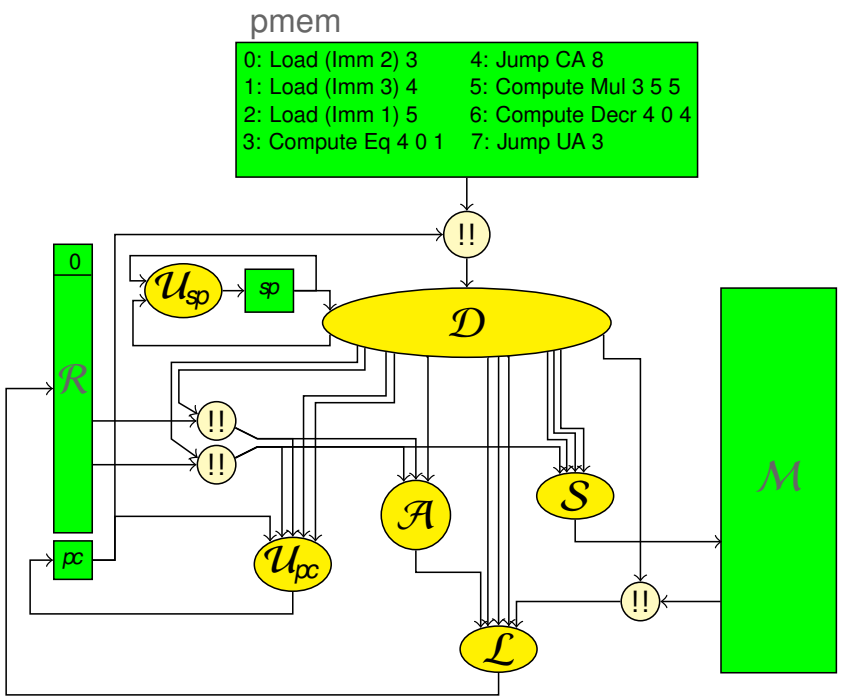
```
spUpd :: SPCode → Int → Int
spUpd spCode sp = case spCode of
                    Up   → incr sp
                    Down → decr sp
                    None → sp
```

Machine code




```
data MachCode
   = MachCode { ldCode      :: LdCode
              , stCode      :: StCode
              , spCode      :: SPCode
              , opCode      :: OpCode
              , immvalueR   :: Int
              , immvalueS   :: Int
              , fromreg0    :: Int
              , fromreg1    :: Int
              , fromaddr    :: Int
              , toreg       :: Int
              , toaddr      :: Int
              , jmpCode     :: JmpCode
              , jumpN       :: Int
              }
```

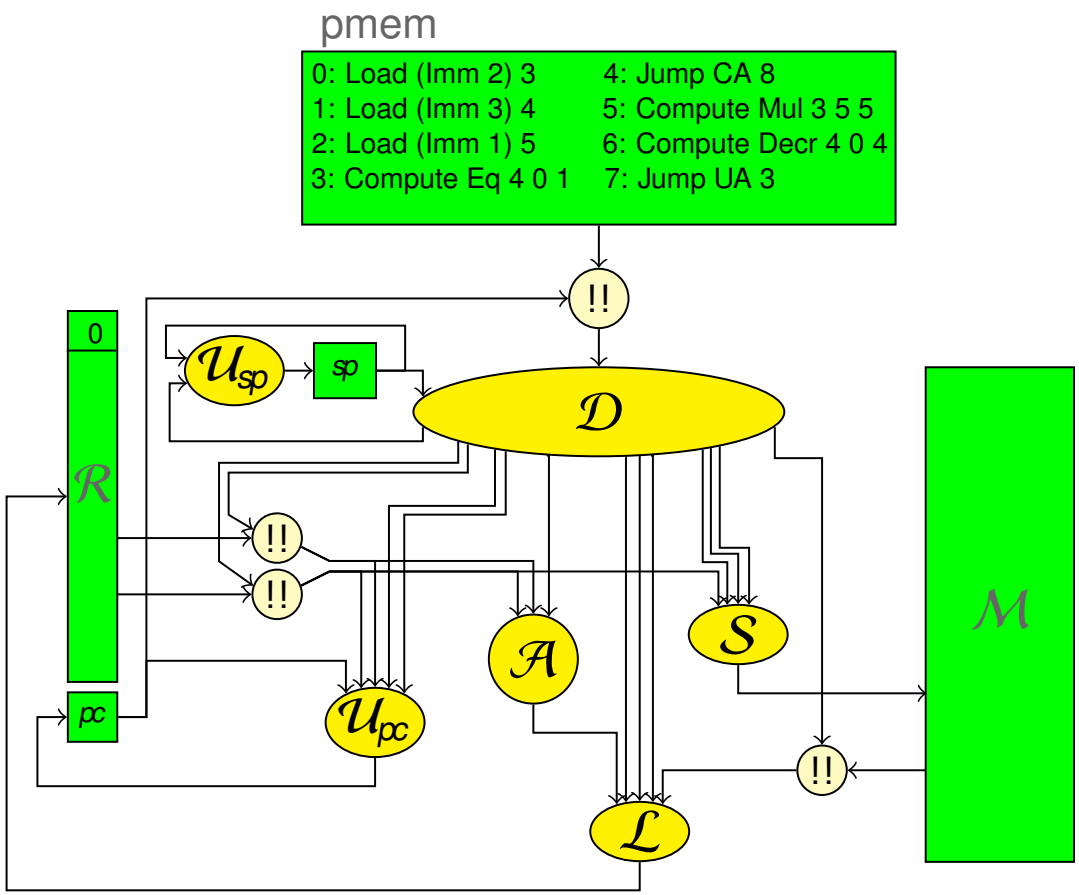
Instruction set



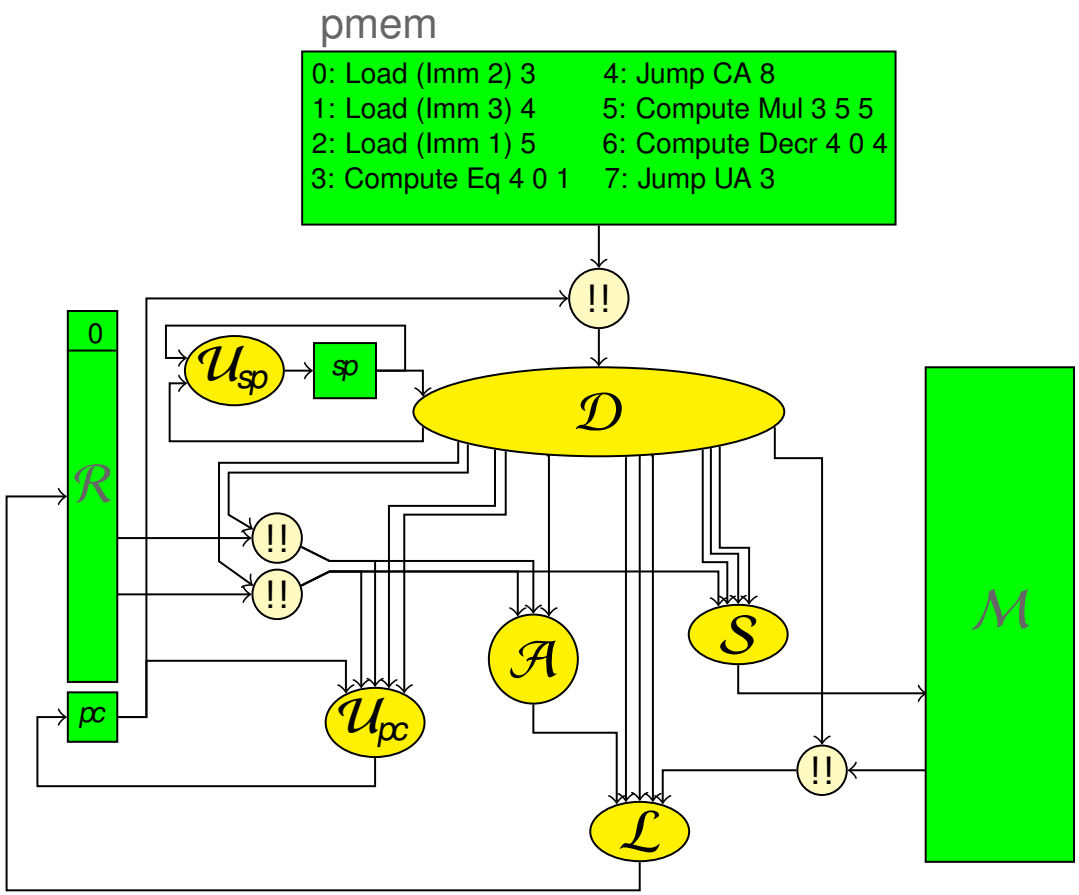
data *Value* = **Addr** *Int* | **Imm** *Int*

data *Assembly*
= **Compute** *OpCode Int Int Int*
| **Jump** *JmpCode Int*
| **Load** *Value Int*
| **Store** *Value Int*
| **Push** *Int*
| **Pop** *Int*

Decoder


pmem
0: Load (Imm 2) 3
1: Load (Imm 3) 4
2: Load (Imm 1) 5
3: Compute Eq 4 0 1
4: Jump CA 8
5: Compute Mul 3 5 5
6: Compute Decr 4 0 4
7: Jump UA 3
!!
0
$\mathcal{U}_{sp}$
sp
$\mathcal{D}$
$\mathcal{R}$
!!
!!
$\mathcal{A}$
$\mathcal{S}$
$\mathcal{M}$
pc
$\mathcal{U}_{pc}$
!!
$\mathcal{L}$

Decoder



Compute *opc* i_0 i_1 i_2

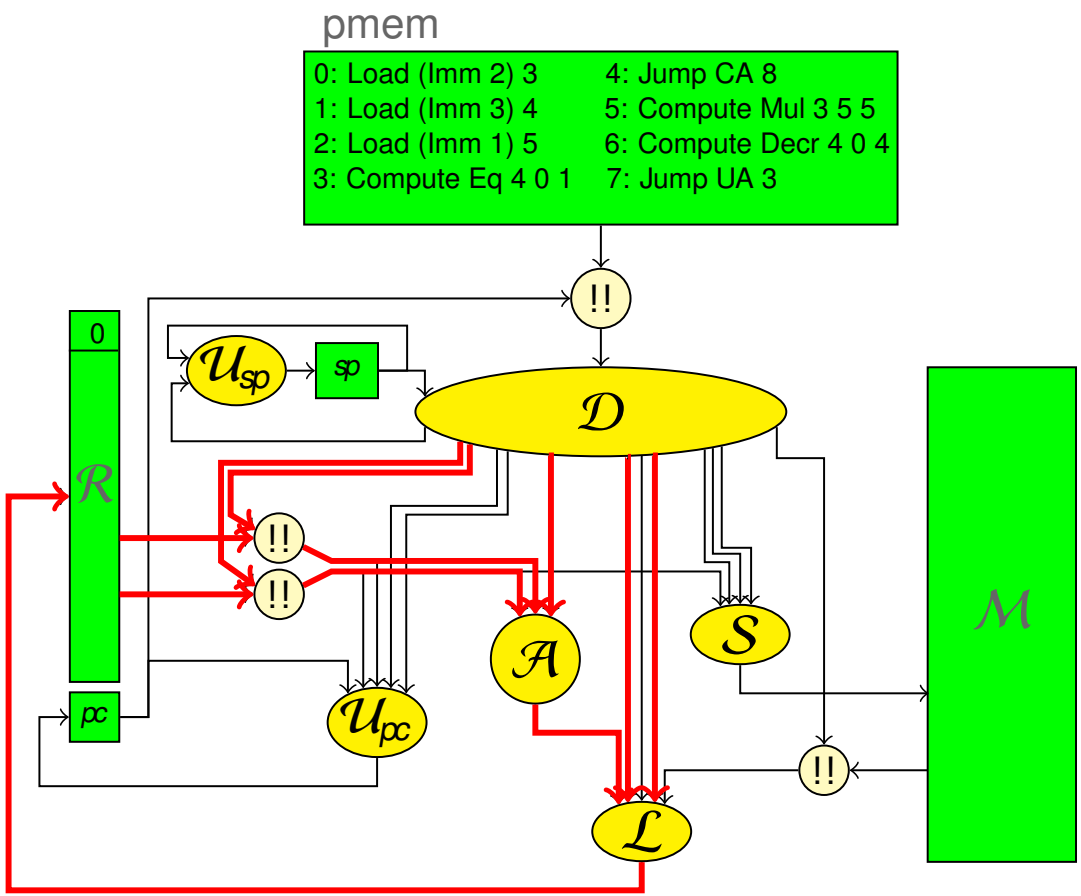
Jump *jc n*

Load (*Imm n*) *j*
Load (*Addr i*) *j*

Store (*Imm n*) *j*
Store (*Addr n*) *j*

Push *i*
Pop *i*

Decoder



$instr =$ **Compute** $opc\ i_0\ i_1\ i_2$

Jump $jc\ n$

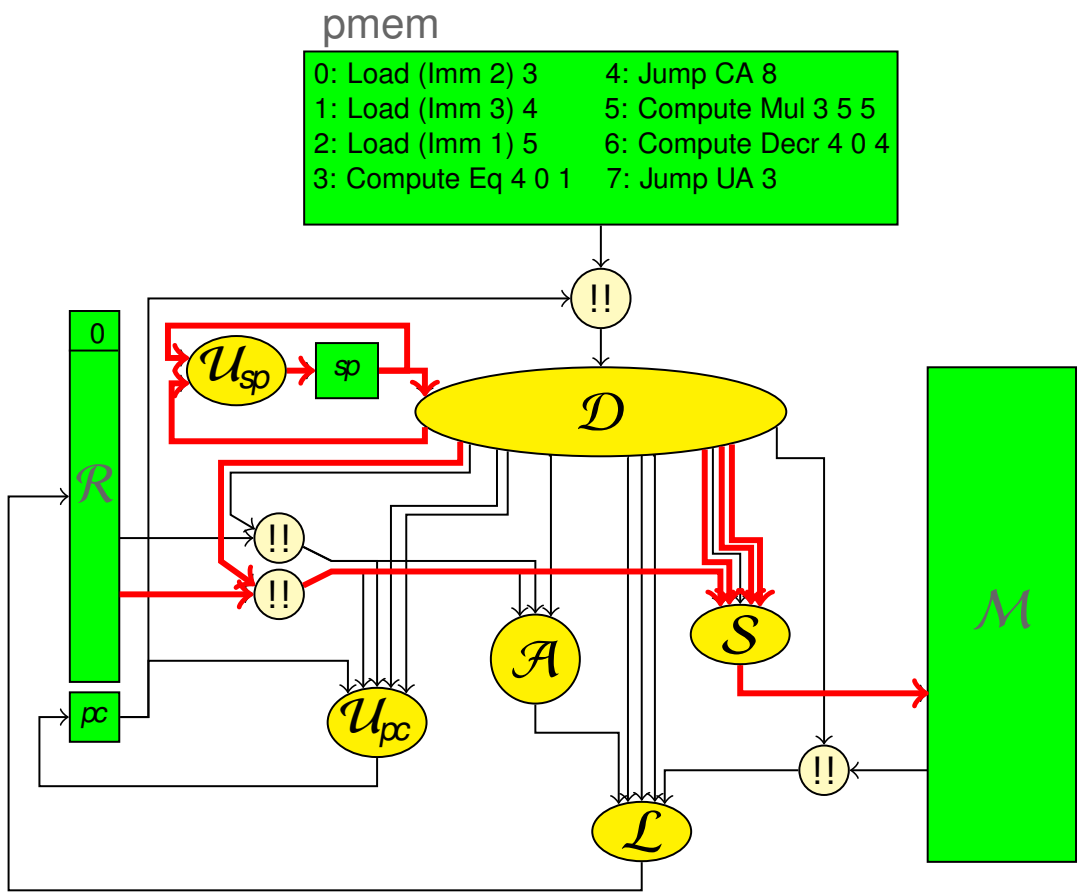
Load $(Imm\ n)\ j$
Load $(Addr\ i)\ j$

Store $(Imm\ n)\ j$
Store $(Addr\ n)\ j$

Push i
Pop i

$$decode\ sp\ instr\ =\ nullcode\{ldCode{=}LdAlu,\ fromreg0{=}i_0,\ fromreg1{=}i_1,\ toreg{=}i_2\}$$

Decoder



Compute *opc* i_0 i_1 i_2

Jump *jc n*

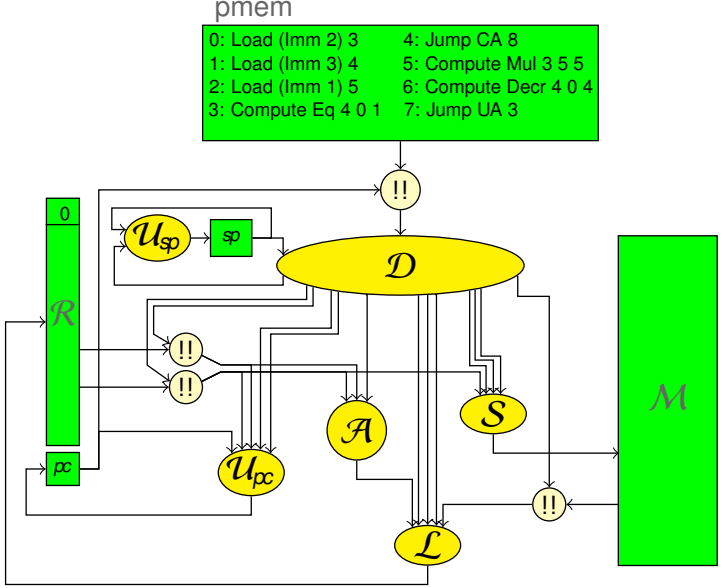
Load (*Imm n*) *j*
Load (*Addr i*) *j*

Store (*Imm n*) *j*
Store (*Addr n*) *j*

instr = **Push** *i*
Pop *i*

decode sp instr = *nullcode*{*stCode*=*StReg*, *spCode*=*Up*,
fromreg0=*i*, *toaddr*=*sp*+1}

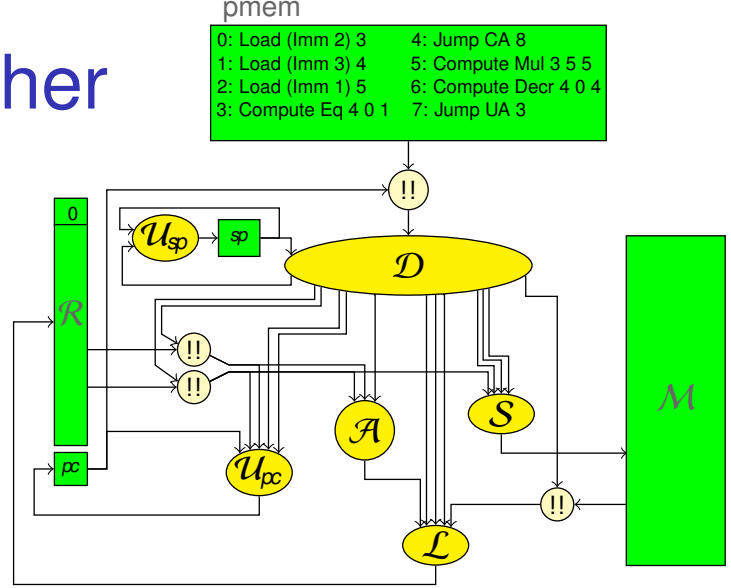
Full decoder



decode sp instr = **case** *instr* **of**

Compute *c i0 i1 i2*	→ *nullcode* {*ldCode* = **LdAlu**,	*opCode* = *c*,	*fromreg0* = *i0*,
		fromreg1 = *i1*,	*toreg* = *i2*}
Jump *jc n*	→ *nullcode* {*jmpCode* = *jc*,	*fromreg0* = *regA*,	*fromreg1* = *jmpreg*,
		jumpN = *n*}	
Load (**Imm** *n*) *j*	→ *nullcode* {*ldCode* = **LdImm**,	*immvalueR* = *n*,	*toreg* = *j*}
Load (**Addr** *i*) *j*	→ *nullcode* {*ldCode* = **LdAddr**,	*fromaddr* = *i*,	*toreg* = *j*}
Store (**Imm** *n*) *j*	→ *nullcode* {*stCode* = **StImm**,	*immvalueS* = *n*,	*toaddr* = *j*}
Store (**Addr** *i*) *j*	→ *nullcode* {*stCode* = **StReg**,	*fromreg0* = *i*,	*toaddr* = *j*}
Push *r*	→ *nullcode* {*stCode* = **StReg**,	*fromreg0* = *r*,	*toaddr* = *incr sp*,
		spCode = **Up**}	
Pop *r*	→ *nullcode* {*ldCode* = **LdAddr**,	*fromaddr* = *sp*,	*toreg* = *r*,
		spCode = **Down**}	

Putting it together



$$
\begin{aligned}
&\mathit{sprockell}\ \mathit{instrs}\ (\mathit{State}\ \{..\}) \quad \_ = \mathit{State}\ \{\mathit{dmem} = \mathit{dmem}', \mathit{regbank} = \mathit{regbank}',\\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pc} = \mathit{pc}', \mathit{sp} = \mathit{sp}'\}\\
&\textbf{where}\\
&\quad \mathit{MachCode}\ \{..\} = \mathit{decode}\ \mathit{sp}\ (\mathit{instrs}\ !!\ \mathit{pc})\\
&\quad \mathit{regbank0} \quad\quad = \mathit{regbank} +\!\!+ [\mathit{pc}]\\
&\quad (x, y) \quad\quad\quad = (\mathit{regbank0}\ !!\ \mathit{fromreg0}, \mathit{regbank0}\ !!\ \mathit{fromreg1})\\
&\quad \mathit{mval} \quad\quad\quad = \mathit{dmem}\ !!\ \mathit{fromaddr}\\
&\quad z \quad\quad\quad\quad = \mathit{alu}\ \ \ \mathit{opCode}\ x\ y\\
&\quad \mathit{regbank}' \quad = \mathit{load}\ \ \ \mathit{regbank}\ \mathit{ldCode}\ \mathit{toreg}\ (\mathit{immvalueR}, \mathit{mval}, z)\\
&\quad \mathit{dmem}' \quad\quad = \mathit{store}\ \ \ \mathit{dmem}\ \mathit{stCode}\ \mathit{toaddr}\ \ (\mathit{immvalueS}, x)\\
&\quad \mathit{pc}' \quad\quad\quad = \mathit{pcUpd}\ (\mathit{jmpCode}, x)\ (\mathit{pc}, \mathit{jumpN}, y)\\
&\quad \mathit{sp}' \quad\quad\quad = \mathit{spUpd}\ \mathit{spCode}\ \mathit{sp}
\end{aligned}
$$

Simulation

$$simulation = selection \,\$\, scanl\ (sprockell\ instrs)\ s0\ clock$$

Fibonacci: Assembly Code

```
[ Load (Imm 5) 5,
  Compute Add 7 5 5,
  Push 5,
  Load (Imm 10) 1,
  Push 1,
  Jump UA 10,
  Store (Addr 1) 0,
  Load (Addr 0) 1,
  WrInstr,
  EndProg,
  Debug "Start func fib",
  Pop 1,
  Store (Addr 1) 1,
  Load (Addr 1) 1,
  Push 1,
  Load (Imm 0) 1,
  Pop 2,
  Compute Equal 2 1 1,
  Jump CR 41,
  Load (Addr 1) 1,
  Push 1,
  Load (Imm 1) 1,
  Pop 2,
  Compute Equal 2 1 1,
  Jump CR 33,
  Load (Addr 1) 2,
  Push 2,
  Load (Imm 9) 5,
  Compute Add 7 5 5,
  Push 5,
  Load (Addr 1) 1,
  Push 1,
  Load (Imm 1) 1,
  Pop 2,
  Compute Sub 2 1 1,
  Push 1,
  Jump UA 10,
  Pop 2,
  Store (Addr 2) 1,
  Push 1,
  Load (Addr 1) 2,
  Push 2,
  Load (Imm 9) 5,
  Compute Add 7 5 5,
  Push 5,
  Load (Addr 1) 1,
  Push 1,
  Load (Imm 2) 1,
  Pop 2,
  Compute Sub 2 1 1,
  Push 1,
  Jump UA 10,
  Pop 2,
  Store (Addr 2) 1,
  Pop 2, ,
  Compute Add 2 1 1,
  Jump UR 2,
  Load (Imm 1) 1,
  Jump UR 2,
  Load (Imm 0) 1,
  Pop 5,
  Jump Back 0   ]
```

To FPGA with CλaSH

- Lists $\Rightarrow$ Vectors,
- Integer types $\Rightarrow$ specific bit widths (Signed or UnSigned),
- Floating point $\Rightarrow$ Fixed-point,

- CλaSH does the encoding of algebraic types, records, etc. into bit vectors ("true machine code"),
- Translates into synthesizable VHDL; then use existing packages to synthesize and put it on FPGA,

- Still under development.

Some more context

- Textual format for (imperative) programming language ⇒ tokenizer, lexer, parse, code generation + simulation of result on Sprockell is a straight pipeline of functions.
- Extended with IO, pointer support,
- Extensions under development: pipelining, caches, networking, multi-core, etc.

Education

No systematic evaluation of educational (dis)advantages, just impressions.

Three groups of people:

- **Computer Science students**: third year undergraduate
- **Embedded Systems students**: master level
- **Lecturers**: compiler construction, concurrency

Undergraduate Computer Science

- Short introduction to the architecture, some demonstration,
- Offered: code of the architecture, plus some function for simulation,
- Assignment: define your own (imperative) programming language, write a compiler (code generation), and simulate the resulting Assembly code,
- Most students have no hardware background, and didn't take the course on compiler construction yet,
- Nevertheless, in two weeks they all manage to complete the task, some even manage to compile and run subroutines,
- Typical reactions: straightforward, transparant, quickly to simulate, easy to understand, Haskell helps to understand processors.

Master program Embedded Systems

- More emphasis on the development of the architecture; comparison with alternative architectures,
- Assignments: extend the architecture with pipelining, IO, cache memory, shared memory system, etc,
- Most students have no background in Functional Programming,
- Nevertheless, most students quickly adopt the notation and simulation,
- Typical reactions: immediate simulation is a huge advantage, you just write down what you think, close resemblance between picture and code, Haskell easier and better understandable than VHDL,
- Follow-up: student projects on non-trivial processors, e.g., MicroBlaze, Xentium, WaveCore (with companies involved).

New developments in educational program

- Integration of three courses into one module, forces lecturers to sit together,
- Relevant module: Functional Programming + Compiler Construction + Concurrent Programming,
- Last two are classical: Java based, ANTLR; Simulation in Java, no direct presence of actual hardware,
- Requests: extend Sprockell, demonstrate it on FPGA.

Thanks

Questions?

`clash.ewi.utwente.nl`