Comparing functional Embedded Domain-Specific Languages for hardware description

João Paulo Pizani Flor

Department of Information and Computing Sciences, Utrecht University

February 13th, 2014

Universiteit Utrecht

Table of Contents

Section 1

Introduction

Universiteit Utrecht

Hardware design

Universiteit Utrecht

Domain-Specific Languages

A computer language (turing-complete or *not*) targeting a *specific application domain.*

Example DSLs:

- SQL (database queries)
- CSS (document formatting)
- MATLAB (Matrix programming)
- VHDL (Hardware description)

Domain-Specific Languages

A computer language (turing-complete or *not*) targeting a *specific application domain.*

Example DSLs:

- SQL (database queries)
- CSS (document formatting)
- MATLAB (Matrix programming)
- VHDL (Hardware description)

A DSL can also be *embedded* in a general-purpose language.

Example EDSLs:

- Boost.Proto (C++ / parser combinators)
- Diagrams (Haskell / programmatic drawing)
- Parsec (Haskell / parser combinators)

Example of an EDSL: Parsec

A simple parser for a "Game of Life"-like input format:

```
dead, alive :: Parser Bool
dead  = fmap (const False) (char '.')
alive = fmap (const True) (char '*')

line :: Parser [Bool]
line  = many1 (dead <|> alive)

board :: Parser [[Bool]]
board = line `endBy1` newline

parseBoardFromFile :: FilePath -> IO [[Bool]]
parseBoardFromFile filename = do
    result <- parseFromFile board filename
    return $ either (error . show) id result
```


Hardware EDSLs

An EDSL used for hardware design-related tasks. Can encompass:

- Modelling / description
- Simulation (validation)
- Formal verification
- Synthesis to other (lower-level) languages

Example of a hardware EDSL

Some Lava code. . .

Section 2

Analyzed EDSLs

Choice criteria

Universiteit Utrecht

Chosen EDSLs

The language we chose to evaluate, with the respective host language, were:

- Lava (Haskell - `chalmers-lava` *dialect*)
- ForSyDe (Haskell)
- Coquet (Coq)

Evaluation criteria

- Simulation
- Verification
- Genericity
- Depth of embedding
- Tool integration
- Extensibility

Section 3

Modeled Circuits

Choice criteria

- Not too simple, not too complex
- Familiar to any hardware designer
  - No signal processing, etc.
- Well-defined, pre-specification
  - Results to verify the models against

Chosen circuits

We cherry-picked circuits from the book "Elements of Computing Systems", as they satisfied all of our demands.

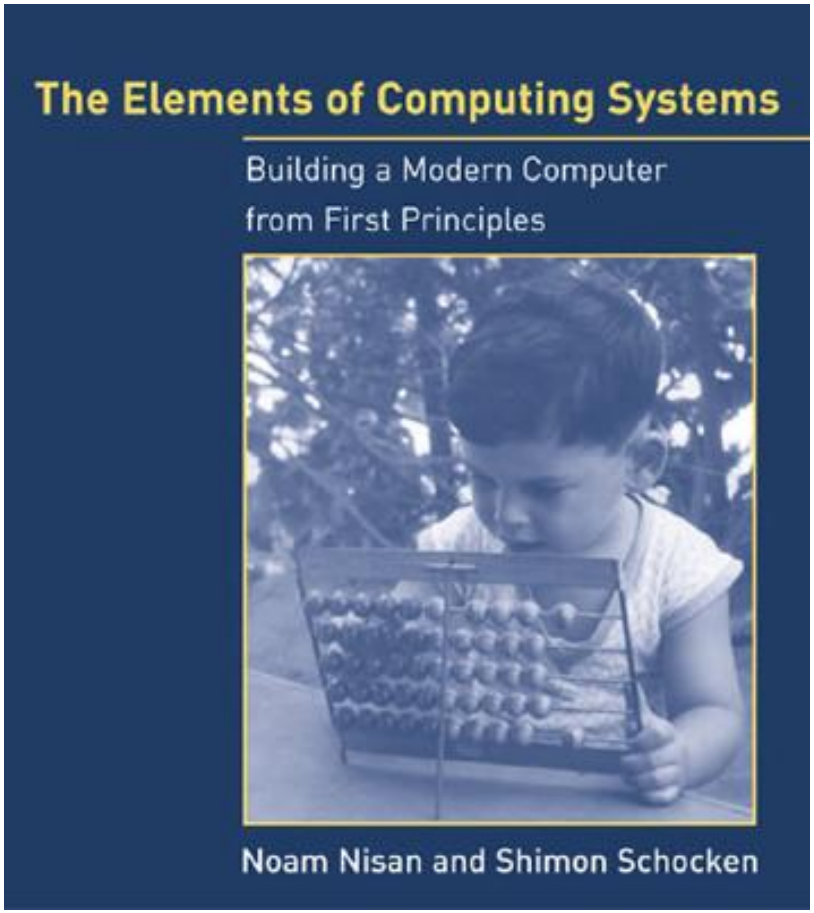


Figure: "Elements of Computing Systems" - Nisan, Schocken, available at `http://www.nand2tetris.org`.

Universiteit Utrecht

Chosen circuits

Circuit 1 A 2-input, 16-bit-wide, simple ALU

Circuit 2 A 64-word long, 16-bit wide memory block

Circuit 3 An *extremely* reduced instruction set CPU, the *Hack* CPU.

Let's take a quick look at each of these circuit's specification. . .

Circuit 1: ALU

Some of the circuit's key characteristics:

- 2 *operand inputs* and 1 *operand output*, each 16-bit wide
- 1 *output flag*
- Can execute 18 different *functions*, among which:
  - Addition, subtraction
  - Bitwise AND / OR
  - Constant outputs
  - Addition of constants to an operand
  - Sign inversion

Circuit 1: block diagram

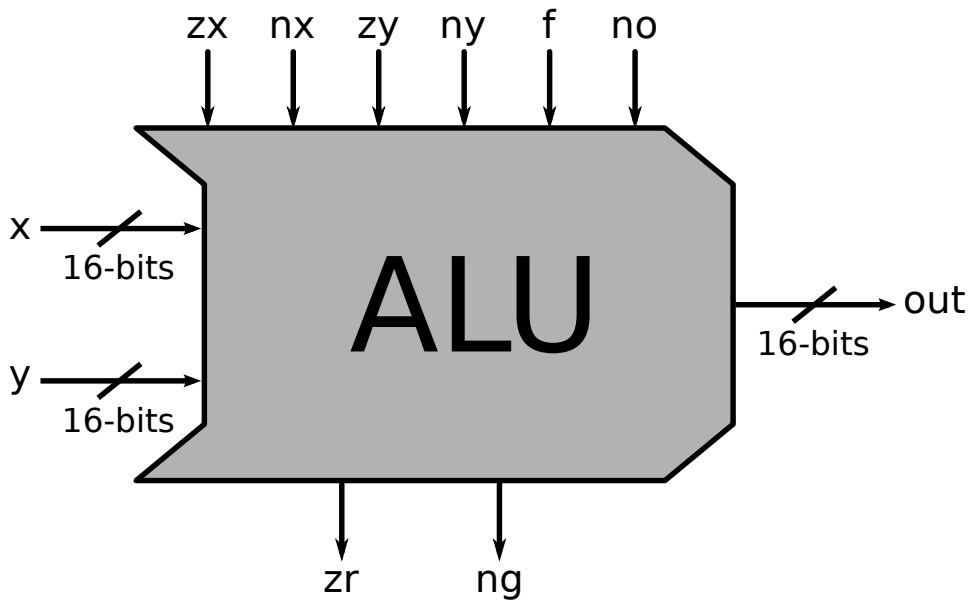


Figure: Input/Output ports of *circuit 1*, the ALU.

Universiteit Utrecht

Circuit 1: specification

The behaviour of the ALU is specified by the values of the *control bits* and *flags*:

- zx and zy — *Zeroes* the "x" and "y" inputs, respectively
- nx and ny — *bitwise negation* on the "x" and "y" inputs
- f — Selects the function to be applied: "f" = 1 for addition, "f" = 0 for bitwise AND
- no — *bitwise negation* on the output ALU output
- zr and ng — The output *flag* "zr" = 1 *iff* the ALU output is zero. "ng" = 1 *iff* the output is negative.

Operations such as bitwise OR, subtraction, etc. can be done by setting the control bits appropriately.

Universiteit Utrecht

Circuit 2: RAM64

Some of the circuit's key characteristics:

- *Sequential* circuit, with clock input
- 64 memory words stored, each 16-bit wide
- Address port has width $\log_2 64 = 16$ bit

Circuit 2: block diagram

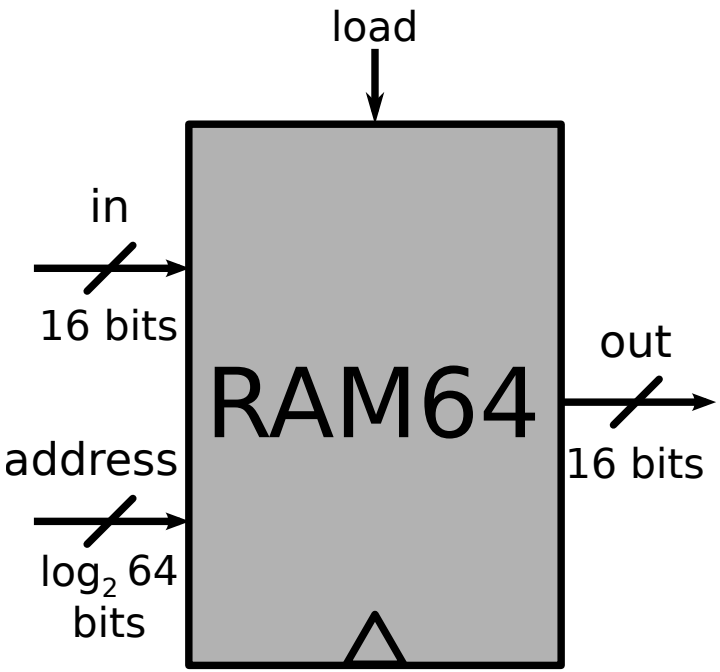


Figure: Input/Output ports of *circuit 2*, the RAM64 block.

Universiteit Utrecht

Circuit 2: specification

The output "out" holds the value at the memory line indicated by "address".

- *Iff* "load" $= 1$, then the value at input "in" will be loaded into memory line "address".
- The loaded value will be emitted on "out" at the *next* clock cycle.

Circuit 3: Hack CPU

- A *very* reduced instruction set CPU
  - Only 2 instructions: "C" and "A"
- Follows the *Harvard architecture*
  - Separate address spaces for *data* and *instruction* memory.
- Instructions are 16-bits wide
  - As well as the memory input and output
- Two *internal* registers: "D" and "A"

Circuit 3: block diagram

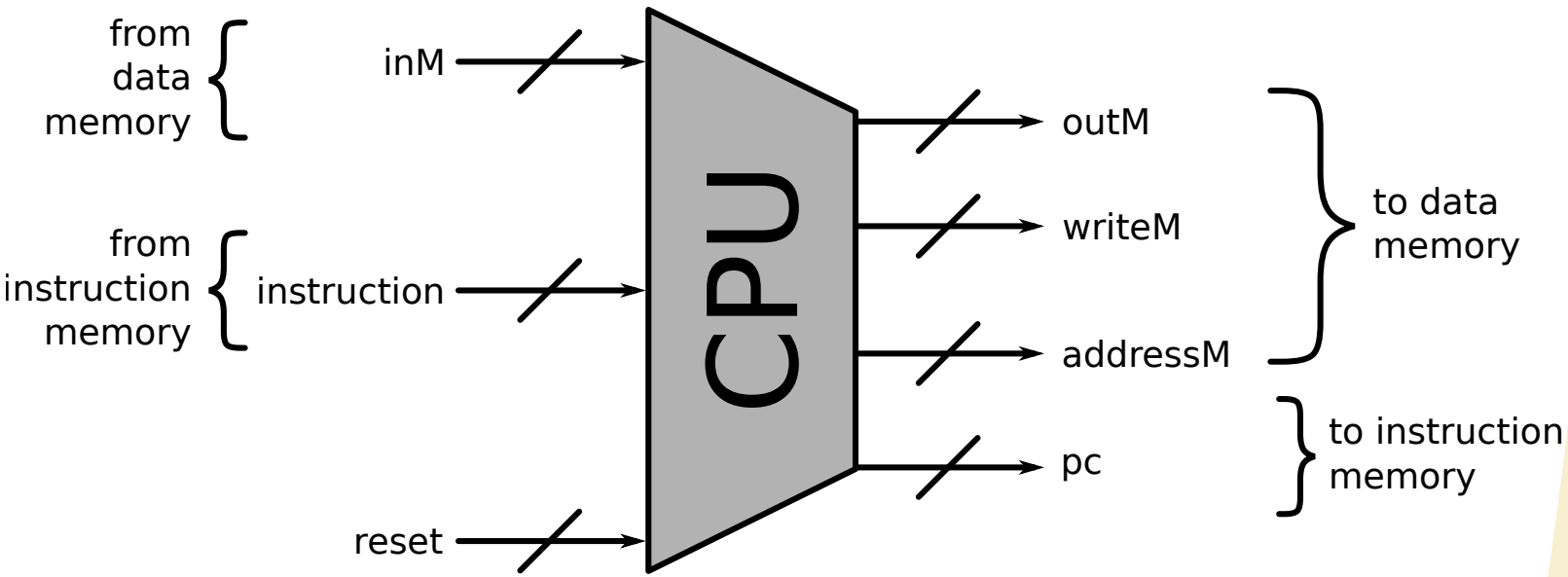


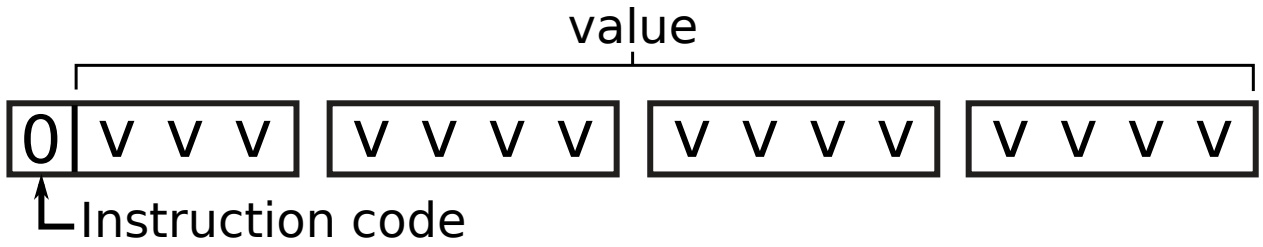
Figure: Input/Output ports of *circuit 3*, the *Hack* CPU.

Universiteit Utrecht

Circuit 3: specification

Circuit 3 runs "A" and "C" instructions, according to the *Hack assembly specification*.

- The "A" instruction: sets the "A" register.



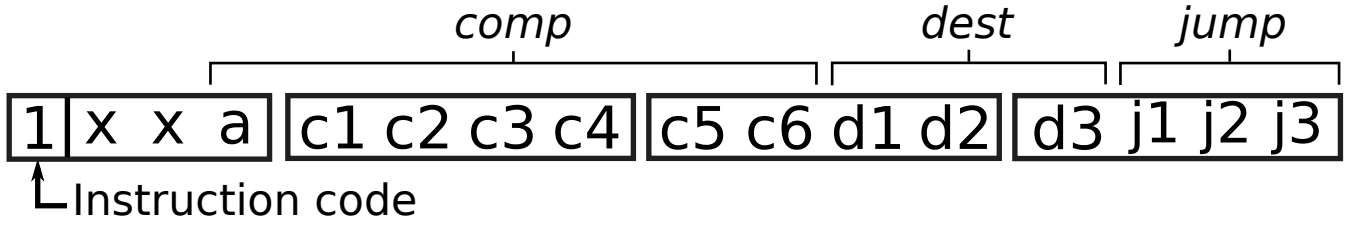
- The value in "A" can be used:
  - As operand for a subsequent computation
  - As address for jumps

Universiteit Utrecht

Circuit 3: specification

Circuit 3 runs "A" and "C" instructions, according to the *Hack assembly specification*.

- The "C" instruction: sets the "C" register, performs *computation* or jumps.



- Some peculiarities:
  - Bits "c1" to "c6" control the ALU
  - *conditional* or *unconditional* jumps
  - *destination* of the computation result: "A", "D", "M"

Universiteit Utrecht

Circuit 3: specification (parts)

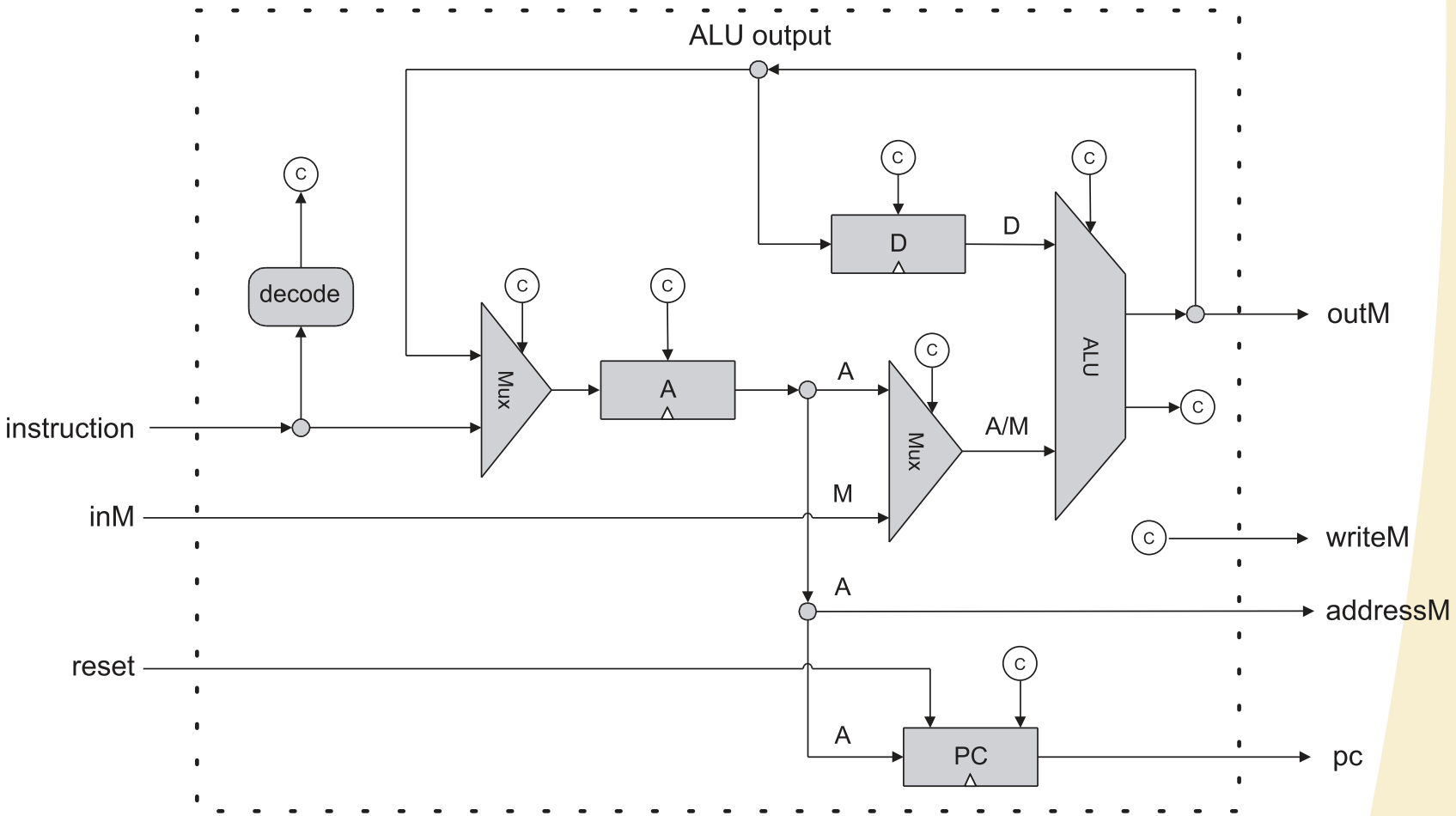


Figure: Parts used to build the *Hack* CPU, and their interconnection.

Universiteit Utrecht

Section 4

Analysis of the EDSLs

Lava

```
type SB = Signal Bool

halfAdder :: (SB, SB) -> (SB, SB)
halfAdder inputs = (xor2 inputs, and2 inputs)

fullAdder :: (SB, (SB, SB)) -> (SB, SB)
fullAdder (cin, (a, b)) = (s, cout)
    where
      (ab, c1) = halfAdder (a, b)
      (s, c2)  = halfAdder (ab, cin)
      cout     = or2 (c1, c2)

rippleCarryAdder :: [(SB, SB)] -> [SB]
rippleCarryAdder ab = s
    where (s, _) = row fullAdder (low, ab)
```


Universiteit Utrecht

Lava

```
type ALUControlBits = (SB, SB, SB, SB, SB, SB)

alu :: ([SB], [SB], ALUControlBits) -> ([SB], SB, SB)
alu (x, y, (zx, nx, zy, ny, f, no)) = (out', zr, ng)
    where x'    = mux (zx, (x, replicate (length x) low))
          x''   = mux (nx, (x', map inv x'))
          y'    = mux (zy, (y, replicate (length x) low))
          y''   = mux (ny, (y', map inv y'))
          out   = let xy'' = zip x'' y''
                  in mux (f, (andl xy'', adder xy''))
          out'  = mux (no, (out, map inv out))
          zr    = foldl (curry and2) low out'
          ng    = equalBool high (last out')
          adder = rippleCarryAdder
```


Universiteit Utrecht

Lava

```
prop_FullAdderCommutative :: (SB, (SB, SB)) -> Signal Bool
prop_FullAdderCommutative (c, (a, b)) =
    fullAdder (c, (a, b)) <==> fullAdder (c, (b, a))
```


Universiteit Utrecht

ForSyDe: ALU

```
bo = bitToBool
bb = boolToBit

aluFunc :: ProcFunc (ALUControl -> WordType -> WordType
                     -> (WordType, ALUFlags))
aluFunc = $(newProcFun [d|
  aluFunc' (zx,nx,zy,ny,f,no) x y =
      ( out,  (bb (out == 0), bb (out < 0)) )
    where
      zf z w   = if bo z then 0 else w
      nf n w   = if bo n then complement w else w
      (xn, yn) = (nf nx $ zf zx $ x,  nf ny $ zf zy $ y)
      out      = nf no $ case f of
                           ALUSum -> xn + yn
                           ALUAnd -> xn .&. yn  |] )
```


Universiteit Utrecht

ForSyDe: Muxes

```
type S a = Signal a
type WT  = WordType

mux2 :: S Bit -> S WT -> S WT -> S WT
mux2 = zipWith3SY "zipWith3SY" $(newProcFun [d|
  f s x y = if s == L then x else y |])

mux2SysDef :: SysDef (S Bit -> S WT -> S WT -> S WT)
mux2SysDef = newSysDef mux2 "mux2Sys"
                       ["sel", "in1", "in2"] ["out"]

mux4 :: S (FSVec D2 Bit) -> S (FSVec D4 WT) -> S WT
mux4 ss is = (mux2' "m1") (sv ! d1) m00 m01 where
  mux2' l = instantiate l mux2SysDef
  sv      = unzipxSY "unzipSel" ss
  iv      = unzipxSY "unzipInp" is
  m00     = (mux2' "m00") (sv ! d0) (iv ! d0) (iv ! d1)
  m01     = (mux2' "m01") (sv ! d0) (iv ! d2) (iv ! d3)

mux4SysDef :: SysDef (S (FSVec D2 Bit) -> S (FSVec D4 WT) -> S WT)
mux4SysDef = newSysDef mux4 "mux4Sys"
                       ["sel", "inputs"] ["out"]
```


Universiteit Utrecht

The Circuit type

```
Context {tech : Techno}
Inductive Circuit : Type -> Type -> Type :=
| Atom : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m},
           techno n m -> Circuit n m

| Plug : forall {n m : Type} {Hfn : Fin n} {Hfm : Fin m}
           (f : m -> n), Circuit n m

| Ser  : forall {n m p : Type},
           Circuit n m -> Circuit m p -> Circuit n p

| Par  : forall {n m p q : Type},
           Circuit n p -> Circuit m q
           -> Circuit (n + m) (p + q)

| Loop : forall {n m p : Type},
           Circuit (n + p) (n + p) -> Circuit n m
```


Universiteit Utrecht

Coquet

```
Definition HADD a b s c: circuit ([:a] + [:b]) ([:s] + [:c]) :=
      Fork2 ([:a] + [:b])
   |> (XOR a b s  &  AND a b c).
```


Universiteit Utrecht

Coquet

```
Program Definition FADD a b cin sum cout :
    circuit  ([:cin] + ([:a] + [:b])) ([:sum] + [:cout]) :=

   (ONE [: cin]  &  HADD a b "s" "co1")
|> Rewire (*  (a, (b,c)) => ((a,b), c)  *)
|> (HADD cin "s" sum "co2" & ONE [: "co1"])
|> Rewire (*  ((a,b), c) => (a, (b,c))  *)
|> (ONE [:sum]  &  OR "co2" "co1" cout).

Next Obligation. revert H; plug_def. Defined.
Next Obligation. plug_auto. Defined.
Next Obligation. revert H; plug_def. Defined.
Next Obligation. plug_auto.Defined.
```


Universiteit Utrecht

Section 5

Conclusions

Results

Universiteit Utrecht

Future work

Thank you!

Questions?

Universiteit Utrecht