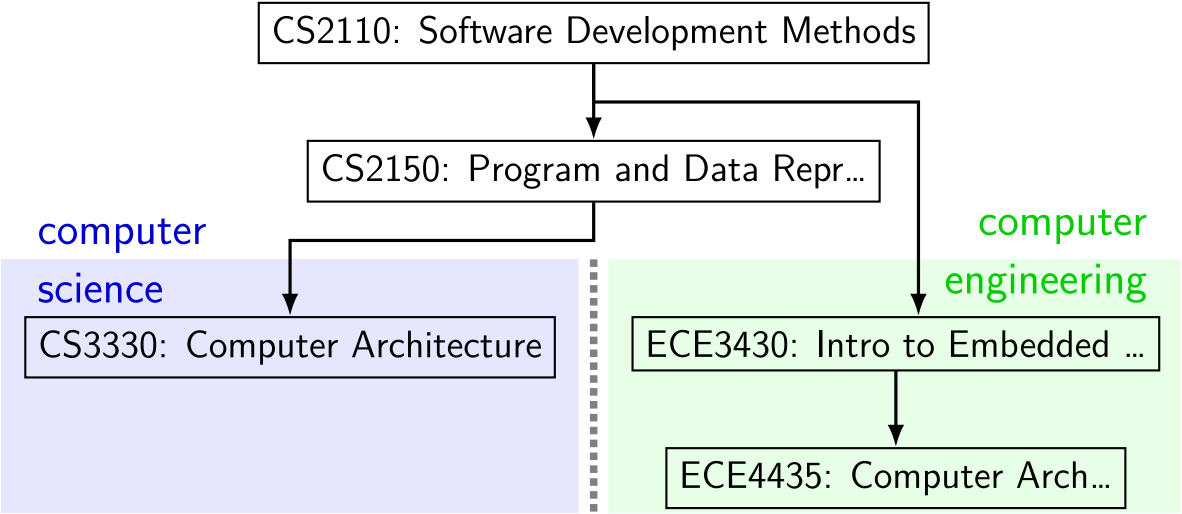
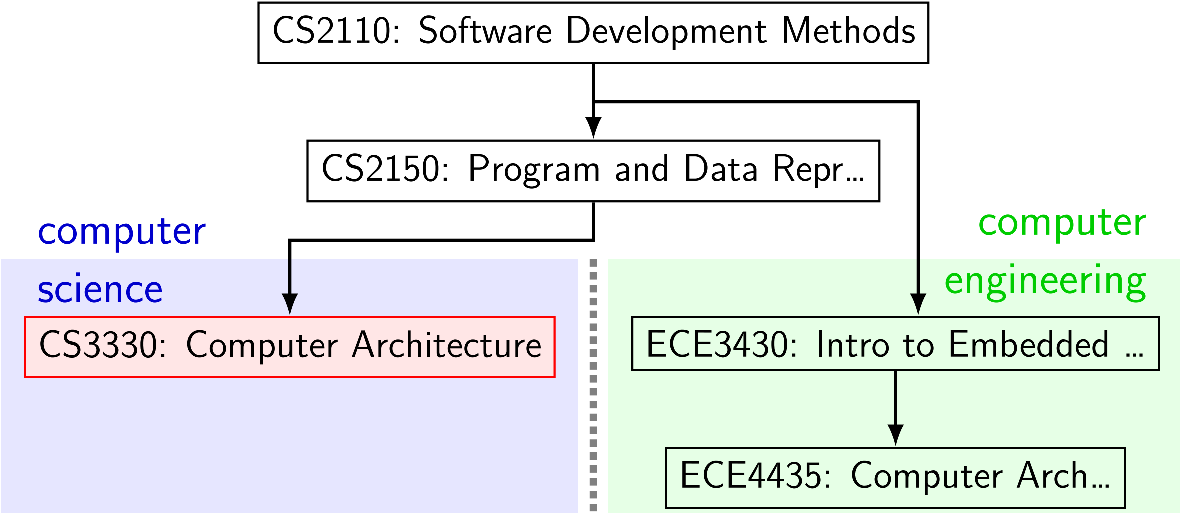
Experiences with a hardware description language in computer-science-focused architecture class

Charles Reiss and Luther Tychnoveich

partial prereq chart



partial prereq chart



two computer arch courses

CS 3330	common topics	ECE 4435
	building processors from simpler components	
	higher-level languages for circuits	
	fetch/execute cycle	
	instruction-level parallelism (ILP)	

two computer arch courses

CS 3330	common topics	ECE 4435
	building processors from simpler components	
	higher-level languages for circuits	
	fetch/execute cycle	
	instruction-level parallelism (ILP)	

objectives well-supported by processor design assignments

two computer arch courses

CS 3330	common topics	ECE 4435
	building processors from simpler components	
	higher-level languages for circuits	
	fetch/execute cycle	
	instruction-level parallelism (ILP)	
skip over some low-level details		full synthesis of CPU
clocking discplines		prep. for real-world chip design
ALU design		important to use industrial HDL
…		
avoid tool-learning overhead		

two computer arch courses

CS 3330		ECE 4435
	common topics	

buildin… h… i…

wanted HDL to support common topics
but didn't want complexity of full HDL
and wanted text-based language
(since students used to text-based programming)

skip over some low-level details	full synthesis of CPU
clocking discplines	prep. for real-world chip design
ALU design	important to use industrial HDL
…	
avoid tool-learning overhead	

HDL assignments: single-cycle

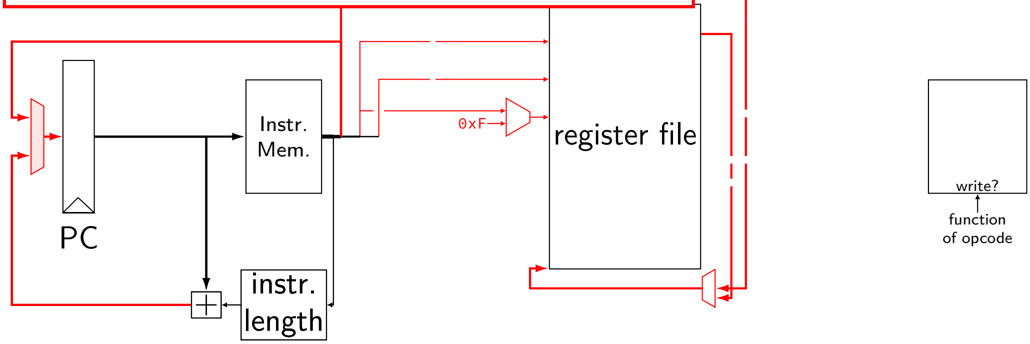
supplied: memories + register file
lab 1: increment PC (var. width instrs)
HW 1: simple register transfer, jump
lab 2: simple arithmetic/conditional move
HW 2: flow control, stack instructions
+ similar sequence for pipelined processors (not shown)

HDL assignments: single-cycle

supplied: memories + register file
lab 1: increment PC (var. width instrs)
HW 1: simple register transfer, jump
lab 2: simple arithmetic/conditional move
HW 2: flow control, stack instructions
+ similar sequence for pipelined processors (not shown)

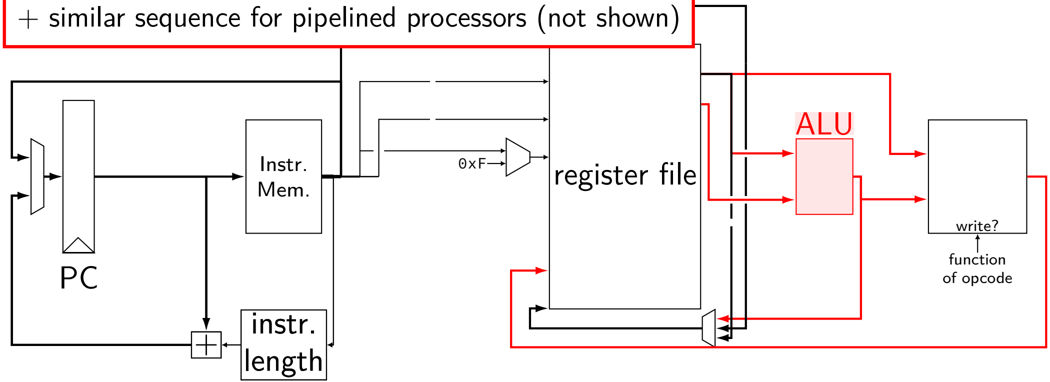
HDL assignments: single-cycle

supplied: memories + register file
lab 1: increment PC (var. width instrs)
HW 1: simple register transfer, jump
lab 2: simple arithmetic/conditional move
HW 2: flow control, stack instructions
+ similar sequence for pipelined processors (not shown)



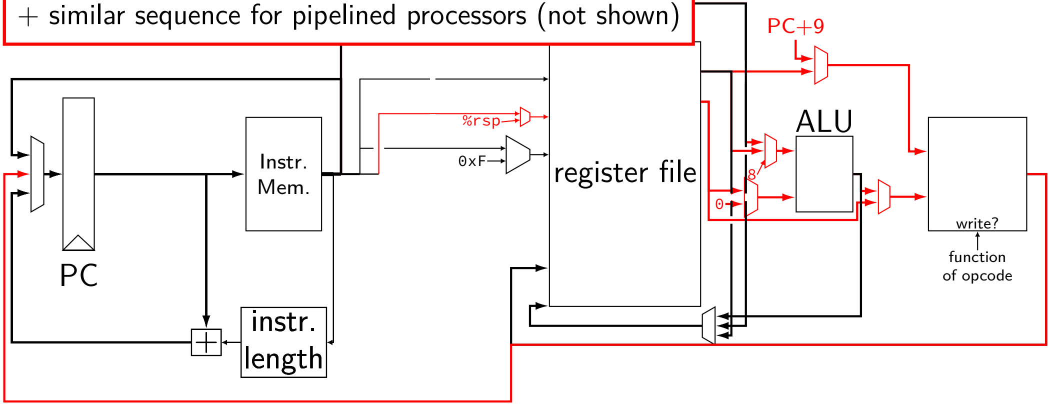
HDL assignments: single-cycle

supplied: memories + register file
lab 1: increment PC (var. width instrs)
HW 1: simple register transfer, jump
lab 2: simple arithmetic/conditional move
HW 2: flow control, stack instructions
+ similar sequence for pipelined processors (not shown)



HDL assignments: single-cycle

supplied: memories + register file
lab 1: increment PC (var. width instrs)
HW 1: simple register transfer, jump
lab 2: simple arithmetic/conditional move
HW 2: flow control, stack instructions
+ similar sequence for pipelined processors (not shown)



CS:APP

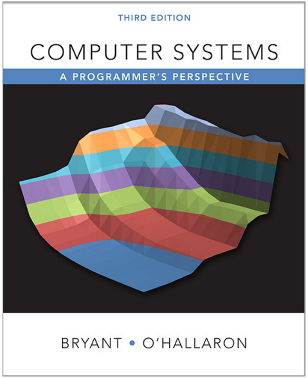
Bryant and O'Hallaron's textbook
seemed to have what we want?

custom hardware description language

    similar goal: teaching software-focused students

processor design described in textbook chapter

    and formally verified
    with Verilog backend option



CS:APP

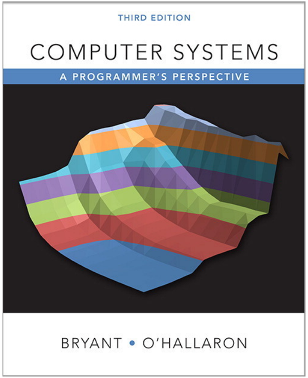
Bryant and O'Hallaron's textbook
seemed to have what we want?

custom hardware description language

  similar goal: teaching software-focused students

processor design described in textbook chapter

  and formally verified
  with Verilog backend option

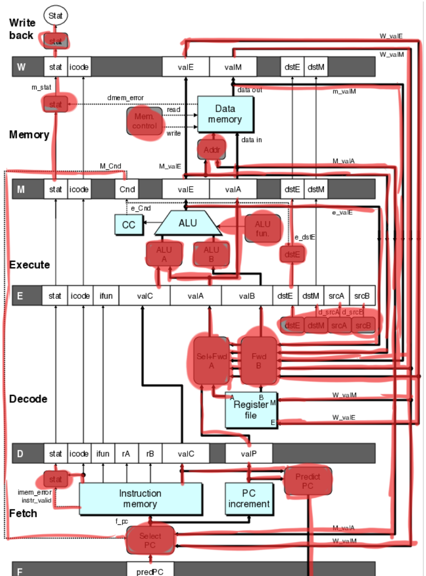


...but no processor construction assignments

  small processor modifications
  not clear these were used actively by authors?

...but lots of built-in components

CS:APP design/built-ins



hilited: implemented in HDL
(rest built-in to simulator)

problem 1: very fixed design
- fixed set of registers
- fixed instruction set
- diff. simulator if diff. pipeline

problem 2: hidden functionality
- machine code parsing
- ALU/memory to register connection

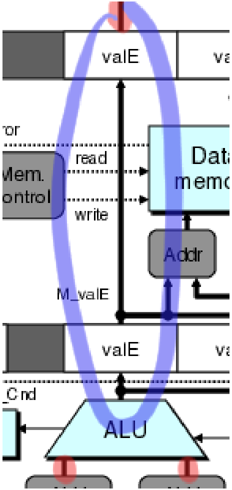
example: hidden built-ins

ALU → pipeline regs → register file input

textbook HDL:

```
int aluA = ...;
int aluB = ...;
int aluOp = ...;
```

```
/* W_valE = pipeline register output */
int w_valE = W_valE;
```

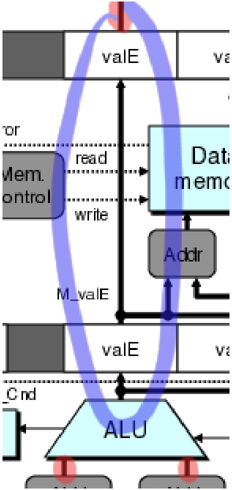


example: hidden built-ins

ALU → pipeline regs → register file input

textbook HDL:

```
int aluA = ...;
int aluB = ...;
int aluOp = ...;
MISSING: pipeling register input
         set from ALU
MISSING: 2nd pipeline register input
         set from piepline register output
/* W_valE = pipeline register output */
int w_valE = W_valE;
NOT CLEAR: w_valE is register file input
```



fixed? hidden-built-ins

```
e_valE = [
    aluOp_is_add : aluA + aluB ;
    ...
];
register eM {
    /* defines e_valE, M_valE */
    valE : 64 = 0;
}
m_valE = M_valE;
register mW {
    /* defines m_valE, W_valE */
    valE : 64 = 0;
}
reg_dstE = W_valE;
```

fixed? hidden-built-ins

```
e_valE = [
    aluOp_is_add : aluA + aluB ;
    ...
];
register eM {
    /* defines e_valE, M_valE */
    valE : 64 = 0;
}
m_valE = M_valE;
register mW {
    /* defines m_valE, W_valE */
    valE : 64 = 0;
}
reg_dstE = W_valE;
```

ALU operations written out by student code

no implicit pipeline register connection

fixed? hidden-built-ins

```
e_valE = [
    aluOp_is_add : aluA + aluB ;
    ...
];
register eM {
    /* defines e_valE, M_valE */
    valE : 64 = 0;
}
m_valE = M_valE;
register mW {
    /* defines m_valE, W_valE */
    valE : 64 = 0;
}
reg_dstE = W_valE;
```

pipeline registers defined by student code

fixed? hidden-built-ins

```
e_valE = [
    aluOp_is_add : aluA + aluB ;
    ...
];
register eM {
    /* defines e_valE, M_valE */
    valE : 64 = 0;
}
m_valE = M_valE;
register mW {
    /* defines m_valE, W_valE */
    valE : 64 = 0;
}
reg_dstE = W_valE;
```

register file signals named `reg_`

avoid confusion with pipeline register signals

output aimed at processor simulation

```
+-------------------- between cycles    0 and    1 ----------------------+
| RAX:                 0   RCX:                0   RDX:                0 |
| RBX:                 0   RSP:                0   RBP:                0 |
| RSI:                 0   RDI:                0   R8:                 0 |
| R9:                  0   R10:                0   R11:                0 |
| R12:                 0   R13:                0   R14:                0 |
| register pP(N) { thePc=0000000000000000 }                              |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+------------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
.-------------------- between cycles    1 and    2 ----------------------+
...
```

output aimed at processor simulation

```
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                0   RCX:                0   RDX:                0 |
| RBX:                0   RSP:                0   RBP:                0 |
| RSI:                0   RDI:                0   R8:                 0 |
| R9:                 0   R10:                0   R11:                0 |
| R12:                0   R13:                0   R14:                0 |
| register pP(N) { thePc=0000000000000000 }                             |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f    |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00    |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00       |
+-----------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
.------------------- between cycles    1 and    2 ----------------------+
...
```

built-in register values (permits easy testing)

output aimed at processor simulation

```
+-------------------- between cycles    0 and    1 ----------------------+
| RAX:                 0   RCX:                 0   RDX:                 0 |
| RBX:                 0   RSP:                 0   RBP:                 0 |
| RSI:                 0   RDI:                 0   R8:                  0 |
| R9:                  0   R10:                 0   R11:                 0 |
| R12:                 0   R13:                 0   R14:                 0 |
| register pP(N) { thePc=0000000000000000 }                               |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f     |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00     |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00        |
+-------------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
.-------------------- between cycles    1 and    2 ----------------------+
...
```

built-in register values (permits easy testing)

decoded instruction for debugging (using textbook's ISA)

output aimed at processor simulation

```
+------------------- between cycles    0 and    1 ----------------------+
| RAX:                  0   RCX:                  0   RDX:                  0 |
| RBX:                  0   RSP:                  0   RBP:                  0 |
| RSI:                  0   RDI:                  0   R8:                   0 |
| R9:                   0   R10:                  0   R11:                  0 |
| R12:                  0   R13:                  0   R14:                  0 |
| register pP(N) { thePc=0000000000000000 }                                 |
| used memory:   _0 _1 _2 _3  _4 _5 _6 _7   _8 _9 _a _b  _c _d _e _f        |
|  0x0000000_:   10 70 13 00  00 00 00 00   00 00 70 1c  00 00 00 00        |
|  0x0000001_:   00 00 00 70  0a 00 00 00   00 00 00 00  10 10 00           |
+-------------------------------------------------------------------------+
pc = 0x0; loaded [10 : nop]
.------------------- between cycles    1 and    2 ----------------------+
...
```

built-in register values (permits easy testing)

decoded instruction for debugging (using textbook's ISA)

contents of user-defined registers

with debugging info

```
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of inputs to built-in components:
pc                     0x0000000000000000
Stat                                  0x1

Values of outputs of built-in components:
i10bytes           0x00000000000000137010

Values of register bank signals:
P_thePc                0x0000000000000000
p_thePc                0x0000000000000001

Values of other wires:
dest                   0x0000000000001370
icode                                 0x1
valP                   0x0000000000000001
```

with debugging info

```
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of inputs to built-in components:
pc                     0x0000000000000000
Stat                                  0x1

Values of outputs of built-in components:
i10bytes           0x00000000000000137010

Values of register bank signals:
P_thePc                0x0000000000000000
p_thePc                0x0000000000000001

Values of other wires:
dest                   0x0000000000001370
icode                                 0x1
valP                   0x0000000000000001
```

signal values for each cycle

with debugging info

```
i10bytes set to 0x137010 (reading 10 bytes from memory at pc=0x0)
pc = 0x0; loaded [10 : nop]
Values of inputs to built-in components:
pc                    0x0000000000000000
Stat                                 0x1

Values of outputs of built-in components:
i10bytes          0x00000000000000137010

Values of register bank signals:
P_thePc               0x0000000000000000
p_thePc               0x0000000000000001

Values of other wires:
dest                  0x0000000000001370
icode                                0x1
valP                  0x0000000000000001
```

signal values for each cycle

actions taken by built-in components

changes over time

first CS:APP offerings (2014)

- used CS:APP's language

initial version (2015-2017)

- targeted for processor construction assignments
- some built-in components + CPU-specific debugging
- otherwise supported multiple processor designs in one simulator

rewrite (2017-2023)

- better error messages
- strict value width enforcement
- avoid floating/default values

changes over time

first CS:APP offerings (2014)

- used CS:APP's language

initial version (2015-2017)

- targeted for processor construction assignments
- some built-in components + CPU-specific debugging
- otherwise supported multiple processor designs in one simulator

rewrite (2017-2023)

- better error messages
- strict value width enforcement
- avoid floating/default values

a common student error

two natural decisions:

machine code: bits 12–16 contain RA (register index)

```
rA = instr_mem_output[12..16];
```

code for “add RA, RB”

```
reg_inputE = rA + rB;
```

a common student error

two natural decisions:

machine code: bits 12–16 contain RA (register index)

```
rA = instr_mem_output[12..16];
```

code for "add RA, RB"

```
reg_inputE = rA + rB;
```

oops: rA is index of RA, not value from register file

"type error": mixing up register index + value

types?

don't want full type system

wires in hardware don't "know" what type they are

solution: check for widths (number of bits) matching

conversion between widths needs to be explicit

wire widths

```
reg_inputE = rA + rB;
```

```
error: Mismatched wire widths.
       The wire 'reg_inputE' is declared as 64 bits wide.
       But a 4 bit wide value is assigned to it:
     -> test.hcl:5
      |
    5 | reg_inputE = rA + rB;
      |              ^^^^^^^
```

results

students often credited assignments with letting them really understand pipelining

complaints re: workload lower over time

- from improved error reporting/testing support?
- from better TA support?
- minor tweaks to assignment writeups?

improved grades over time

- ...but mostly from autograder availability

made instructors comfortable teaching more advanced ILP (out-of-order)

future

`https://github.com/charlesreiss/hclrs-assignments`

for unrelated reasons, our Comp Arch courses have changed

no longer split comp sci/computer engineering

less detailed pipelining/HDL coverage in core

our HDL

multi-bit *signals*

"connections" via assignment syntax

- signalName = expression
- expression can include C-like arithmetic

built-in storage components (w/ in+out signals)

declared "register banks" (w/ in+out signals)

case expressions (copied from CS:APP) represents MUXes

```
[ cond1: value1; cond2: value2; ... ]
```

- avoids procedural-like syntax for combinatorial circuits

register banks

```
register xY {
    foo : 64 = 0;
    bar : 64 = 0;
};
```

declares 2 registers ‘foo’, ‘bar’

inputs: `x_foo`, `x_bar`

outputs: `Y_foo`, `Y_bar`

64-bit width, initial value `0`

meant to allow following CS:APP’s naming:

d_foo set by decode stage, E_foo read by execute stage

learning goals

purpose of clock in synchronous digital logic

hardware is inherently parallel

making decisions with multiplexors

critical path length and clock rate

parallelism via pipelining

data and control hazards in a pipelined processor

assignment sequence

week 1–2: single-cycle CPU

- lab 1: incrementing PC (variable-width instruction)
- HW 1: simple register transfer instructions, jump
- lab 2: simple arithmetic/conditional move
- HW 2: flow control, stack instructions

week 3–4: pipelined CPU

- lab 1: two-stage pipeline
- HW 1: five-stage pipeline (hazard handling w/ forwarding)
- lab 2: memory instructions (hazard handling w/ stalling)
- HW 2: rest (branch prediction, more stalling handling)

built-in components

customized to our assignments

two-port main memory

- 80-bit "instruction" read port
- 64-bit "data" read+write port

two read port, two write port register file

- 15 64-bit registers + 1 zero register

"Stat" simulation control signal

- for halting simulator

changes over time

first CS:APP offerings (2014)

- used CS:APP's language

initial version (2015-2017)

- targeted for processor construction assignments
- some built-in components + CPU-specific debugging
- otherwise supported multiple processor designs in one simulator

rewrite (2017-2023)

- better error messages
- strict value width enforcement
- avoid floating/default values

error message refinement

`wire foo = bar;`

(intended to be: "`wire foo : 64; foo = bar;`")

first version: "`syntax error on line XX`"

---

error message refinement

```
wire foo = bar;
```

(intended to be: "`wire foo : 64; foo = bar;`")

first version: "`syntax error on line XX`"

```
error: Wire declaration missing width:
     -> test.hcl:5
      |
    5 | wire foo = bar;
      |      ^^^^^^
error: Wire declaration must be separate from assignment:
     -> test.hcl:5
      |
    5 | wire foo = bar;
      |      ^^^^^^
```

error message fixes

specific generic error messages ("expected one of X, Y, Z after ...")

custom error messages for special cases

parsed common error patterns

- didn't expand HDL language to keep it simple to explain
- ...but did necessary parsing work

changes over time

first CS:APP offerings (2014)

- used CS:APP's language

initial version (2015-2017)

- targeted for processor construction assignments
- some built-in components + CPU-specific debugging
- otherwise supported multiple processor designs in one simulator

rewrite (2017-2023)

- better error messages
- strict value width enforcement
- avoid floating/default values

case expressions

CS:APP syntax for "case expressions"

```
destination = [
        conditionOne : valueWhenConditionOne;
        conditionTwo : valueWhenConditionTwo;
        conditionThree : valueWhenConditionThree;
        conditionFour : valueWhenConditionFour;
];
```

if/else if/else if/else syntax

presented as representing a MUX

problems with default values

```
reg_dstE = [
        icode == OPQ : rB;
        icode == RMMOVQ : REG_NONE;
];
```

problems with default values

```
reg_dstE = [
        icode == OPQ : rB;
        icode == RMMOVQ : REG_NONE;
];
```

bug: case missing

- get some implicit value (`0` in our first tool)

problems with default values

```
reg_dstE = [
        icode == OPQ : rB;
        icode == RMMOVQ : REG_NONE;
];
```

bug: case missing

get some implicit value (`0` in our first tool)

our solution: **required** default:

```
reg_dstE = [
        icode == OPQ : rB;
        icode == RMMOVQ : REG_NONE;
        true : some_value_obvious_in_debugging;
];
```

equality issues?

```
reg_dstE = [
        /* student meant: icode == OPQ : ...*/
        OPQ : rB;
        /* student meant: icode == RMMOVQ : ...*/
        RMMOVQ : REG_NONE;
        true : some_value_obvious_in_debugging;
];
```

bug: missing comparison

non-zero constants (like OPQ) are true (following C semantics)

equality issues?

```
reg_dstE = [
        /* student meant: icode == OPQ : ...*/
        OPQ : rB;
        /* student meant: icode == RMMOVQ : ...*/
        RMMOVQ : REG_NONE;
        true : some_value_obvious_in_debugging;
];
```

bug: missing comparison

non-zero constants (like OPQ) are true (following C semantics)

our solution: **compile error**: "multiple default cases"

changes over time

first CS:APP offerings (2014)

- used CS:APP's language

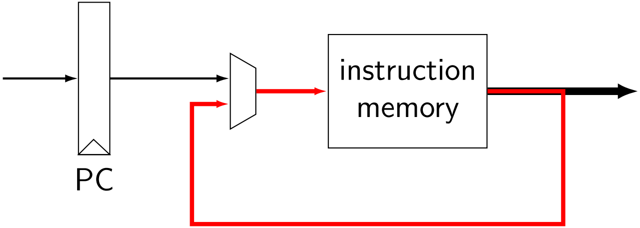
initial version (2015-2017)

- targeted for processor construction assignments
- some built-in components + CPU-specific debugging
- otherwise supported multiple processor designs in one simulator

rewrite (2017-2023)

- better error messages
- strict value width enforcement
- avoid floating/default values

unintended combinatorial circuit loops



very common student error pattern

```
error: Circular dependency detected:
           'pc' depends on 'i10bytes' and
           'i10bytes' depends on 'pc'
```

why not Verilog/VHDL?

generally: less irrelevant-to-us boilerplate

- modules, etc.
- worrying about which clock edge triggers logic
- memory/register file interface details

avoid procedural-looking logic

- "if (a==0) begin ..."
- emphasize parallel nature of hardware

limit to plausible synthesizable logic

avoid implicitly truncated values

following a textbook?

assignments still followed CS:APP textbook's design

built-in components in simulator followed textbook's naming

...but some of those we found less-than-ideal:

- register write enable/reset signal naming
- inconsistent register file/memory interface

textbook register control

`register xY { foo : 64 = 0; }`

register output `Y_foo`

register input `x_foo`

changes Y_foo value next cycle

write disable `stall_Y`

affects Y_foo value next cycle

not only signal needed to implement stalling

reset `bubble_Y`

affects Y_foo value next cycle

textbook register control

`register xY { foo : 64 = 0; }`

register output `Y_foo`

register input `x_foo`

- changes Y_foo value next cycle

write disable `stall_Y`

- affects Y_foo value next cycle
- not only signal needed to implement stalling

reset `bubble_Y`

- affects Y_foo value next cycle

textbook register control

`register xY { foo : 64 = 0; }`

register output `Y_foo`

register input `x_foo`

- changes Y_foo value next cycle

write disable `stall_Y`

- affects Y_foo value next cycle
- not only signal needed to implement stalling

reset `bubble_Y`

- affects Y_foo value next cycle

processor design and HDLs

both courses: computer processor design assignments

use a *hardware description language*

to describe processor + simulate/test it

typical industrial options: Verilog, VHDL, …