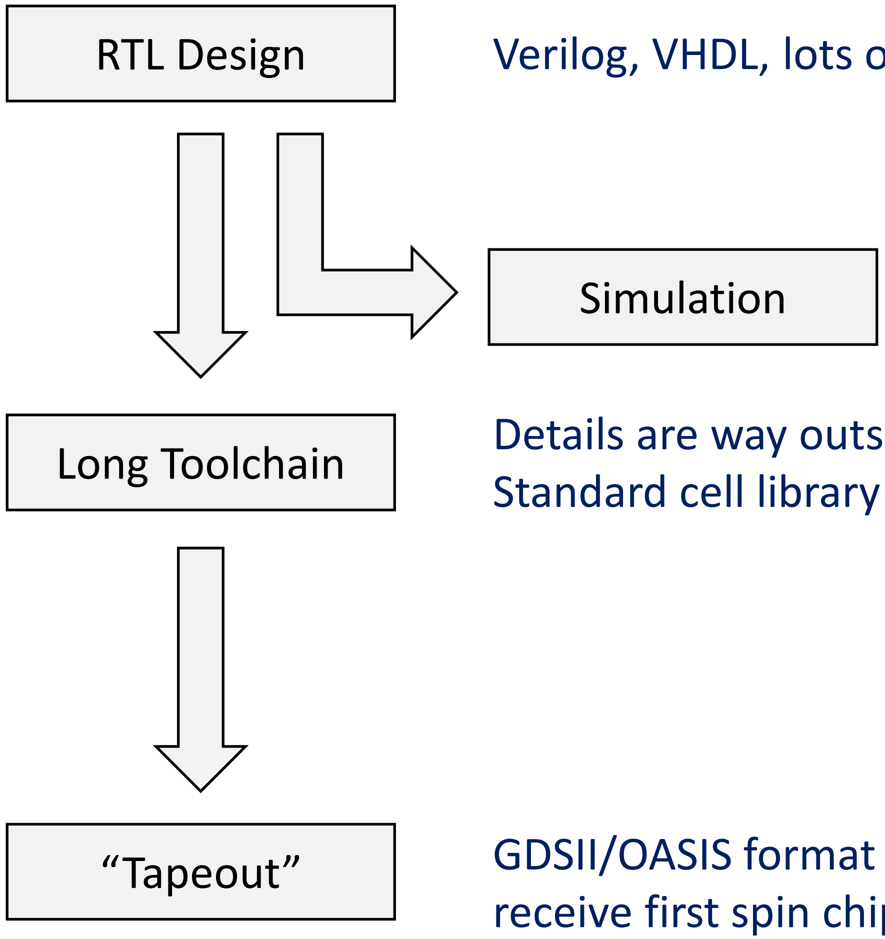
CS152: Computer Systems Architecture Hands-On Processor Development

Sang-Woo Jun

Winter 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Canonical Microprocessor Design Flow

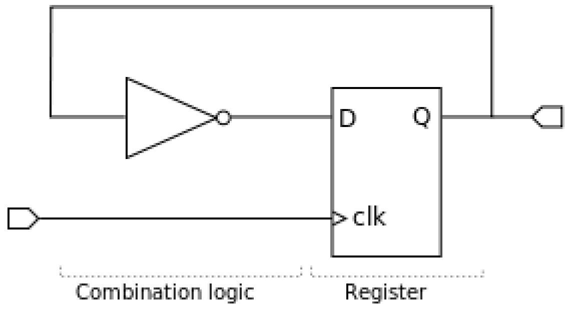


Verilog, VHDL, lots of custom, in-house tools…

Details are way outside scope of cs152
Standard cell library from target foundry/technology is an input

GDSII/OASIS format sent to foundry,
receive first spin chip in a few months

Image source: Alinja, English Wikipedia



RTL (Register-Transfer-Level)

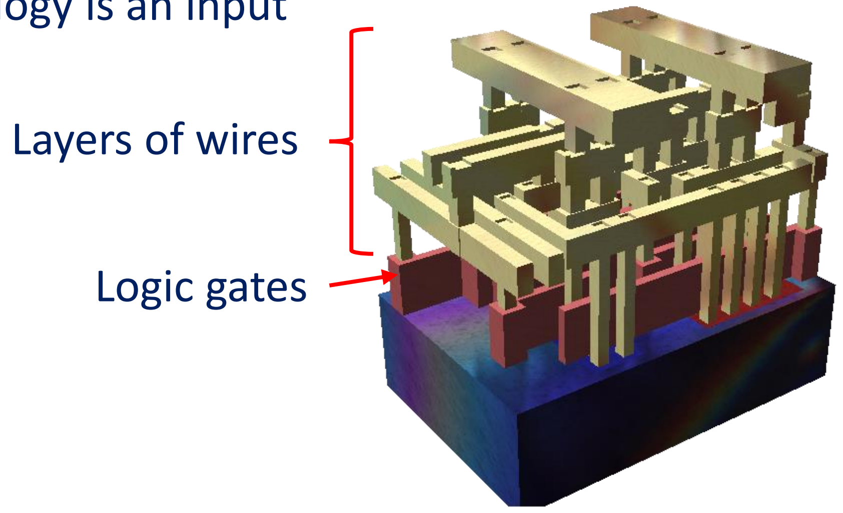
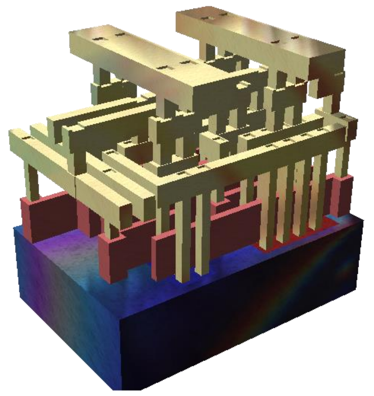


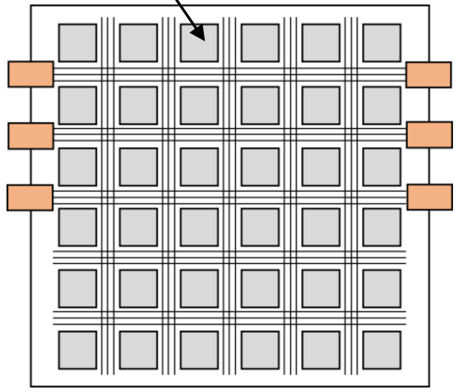
Image source: David Carron, English Wikipedia

Prototyping Using FPGAs

- ***Field-Programmable*** Gate Array
- A grid of "Configurable Logic Blocks" (CLB)
  - Each CLB can be programmed to act like logic gates (stores truth table)
  - A flexible on-chip network can act like wires
- Can be reconfigured in seconds
- CLBs and on-chip network emulating actual silicon
  - Not as dense, not as fast
  - Great for prototyping!



"Configurable logic block (CLB)"



Toolchains for FPGA development

- Typically vendor-specific
  - Xilinx: Vivado, Vitis
  - Intel/Altera: Quartus
  - Lattice: Diamond
- Robust open-source projects
  - Yosys, nextpnr, arachnepnr, icestorm, …
  - Mostly centered around low-power Lattice FPGAs
  - We will use this!

High-Level Hardware-Description Languages

- Modern circuit design is aided heavily by Hardware-Description Languages
  - Relatively high-level description to compiler
  - Toolchain performs “synthesis”, translating them into gates, also place, route, etc
  - High-end chips require human intervention in each stage for optimization
- Wide spectrum of languages and tools
  - Register-Transfer-Level (RTL) languages: Verilog, VHDL, … Efficient, difficult to program
    - Registers (state), and combinational logic
  - “High-Level Synthesis”: Uses familiar software programming languages
    - C-to-gates, OpenCL, … Easy to program, inefficient
    - Typically compiles to Verilog/VHDL

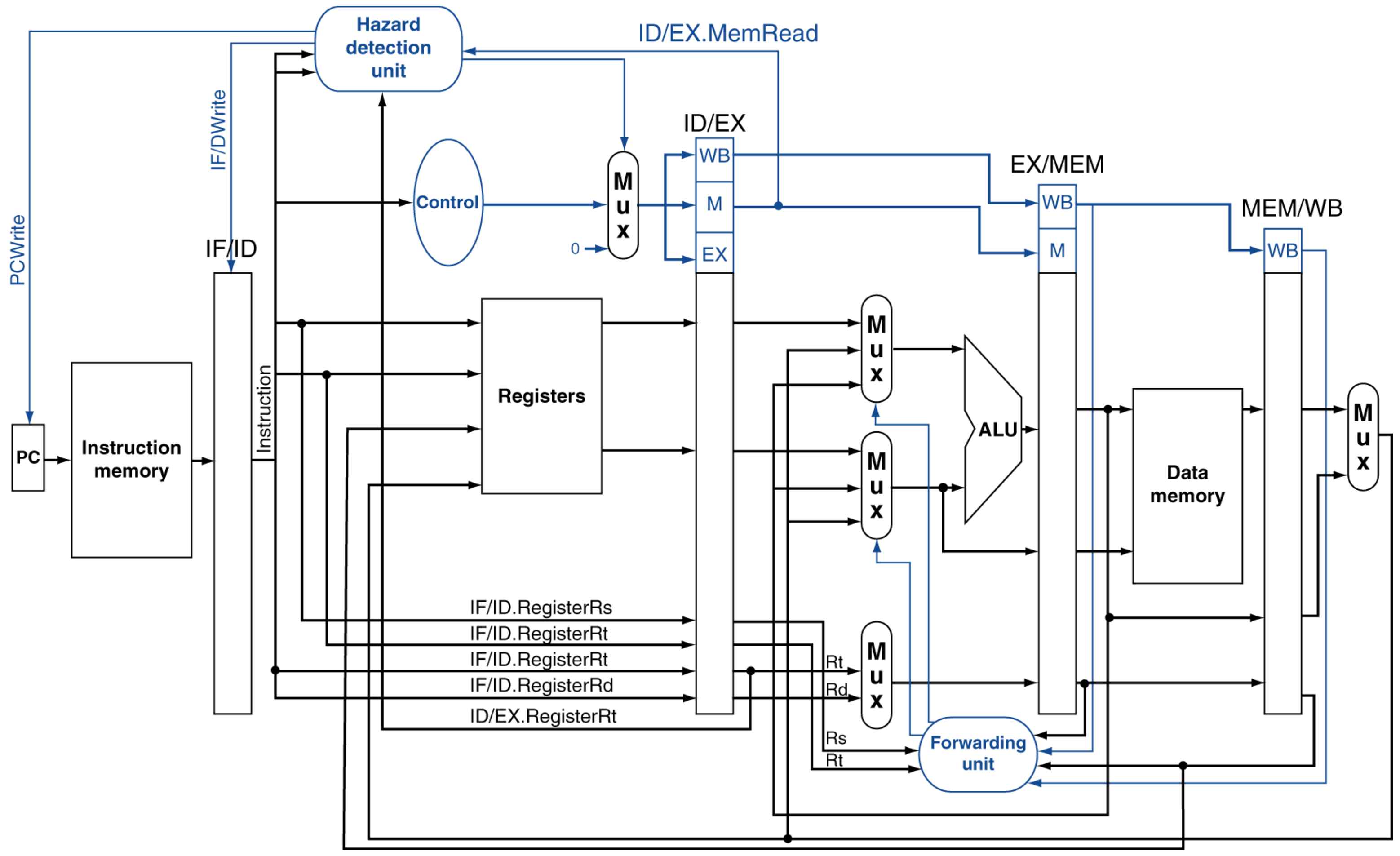
Bluespec System Verilog (BSV)

- "High-level HDL without performance compromise"
- Comprehensive type system and type-checking
  - Types, enums, structs
- Static elaboration, parameterization (Kind of like C++ templates)
  - Efficient code re-use
- Efficient functional simulator (bluesim) printf's and user input during simulation!
- Most expertise transferrable between Verilog/Bluespec

In a comparison with a 1.5 million gate ASIC coded in Verilog, Bluespec demonstrated a 13x reduction in source code, a 66% reduction in verification bugs, equivalent speed/area performance, and additional design space exploration within time budgets.

-- PineStream consulting group

Low-level control flow design



Not very intuitive… We will revisit with code later

Hands-On Processor Development

- We will experience the impact of ideas we cover
  - Using synthesizable processor implementation in Bluespec
  - Synthesized for an FPGA using open-source tools

- “How does this change effect the critical path?”
- “How does this change effect the cycle count?”
- “How does this change effect chip resource utilization?”

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

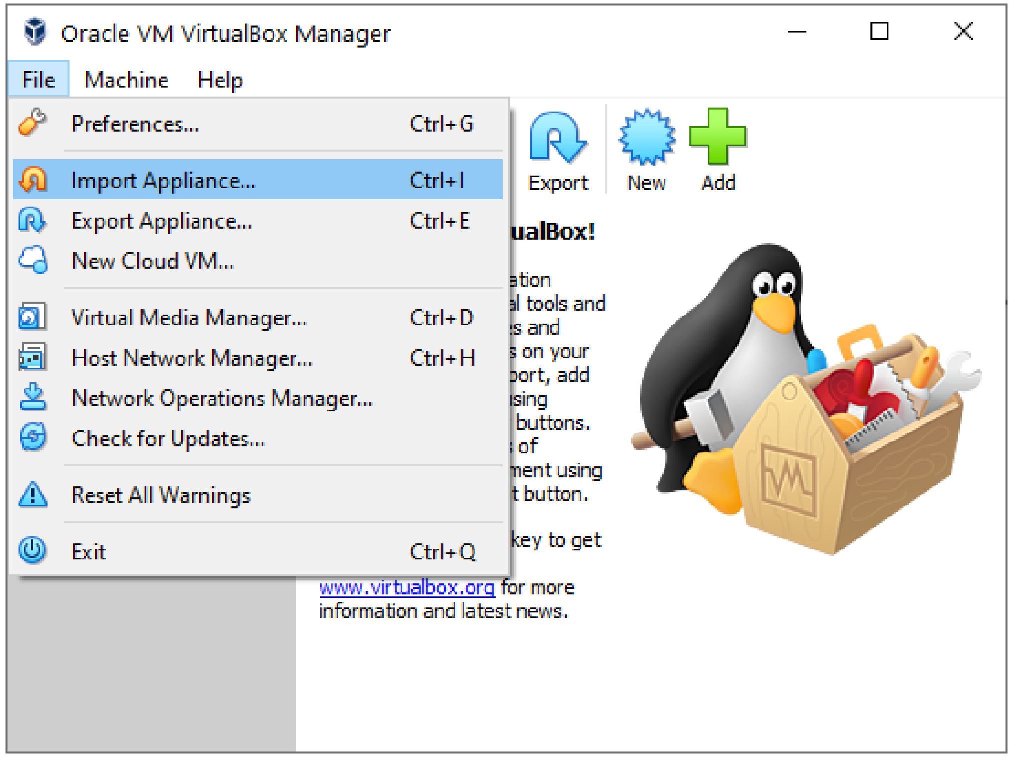
Getting Started

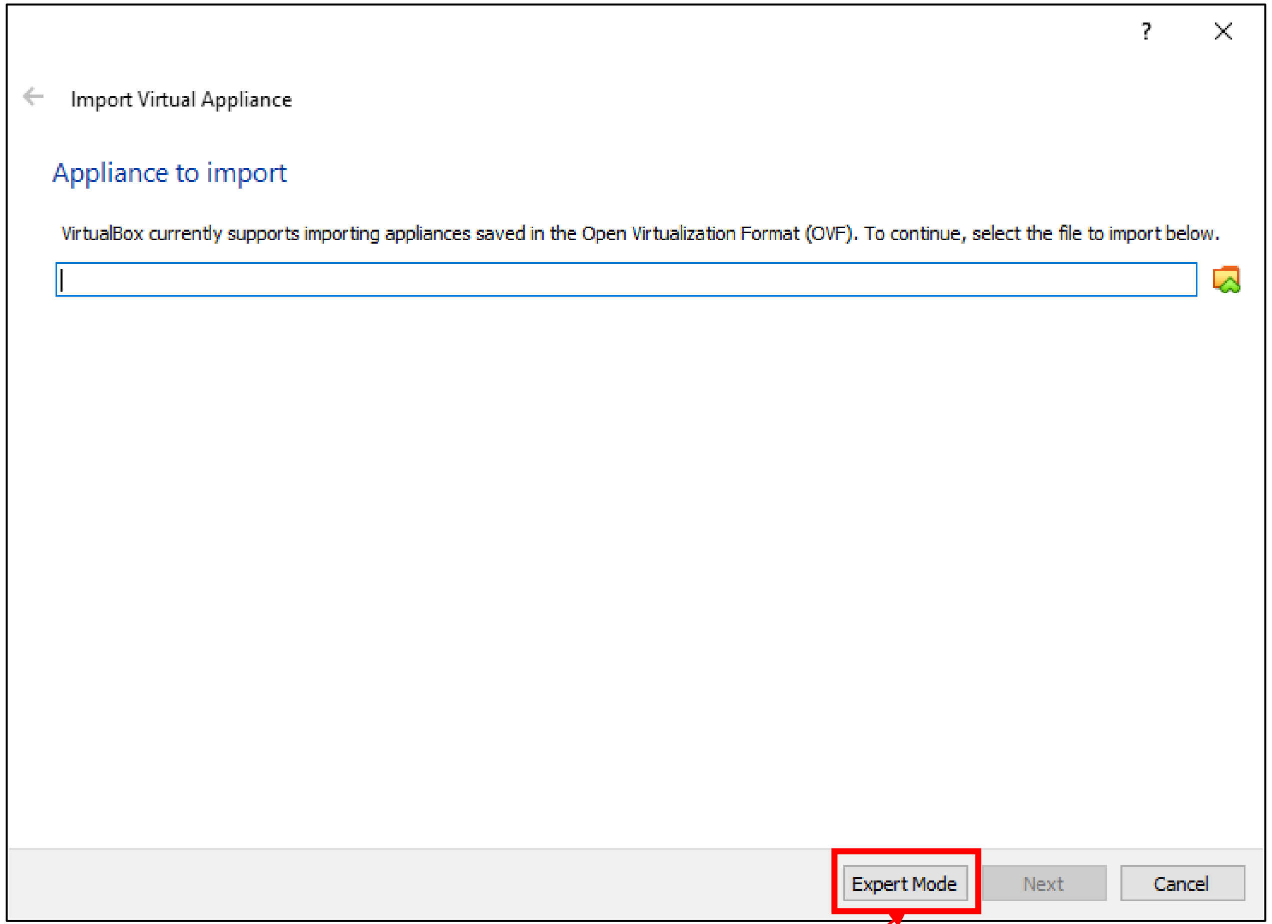
- Virtual machine with all tools installed, available at:
  - cs152-ubuntu.ova (4 GB!)

    https://drive.google.com/file/d/1ia-u3XWJ08EQI6KZEykJhkEd4Htt2tAz/view?usp=sharing
- First, install Oracle Virtualbox
  - Open-source virtual machine
  - High performance with minimal configuration

Getting Started

- ❑ Import the downloaded VM

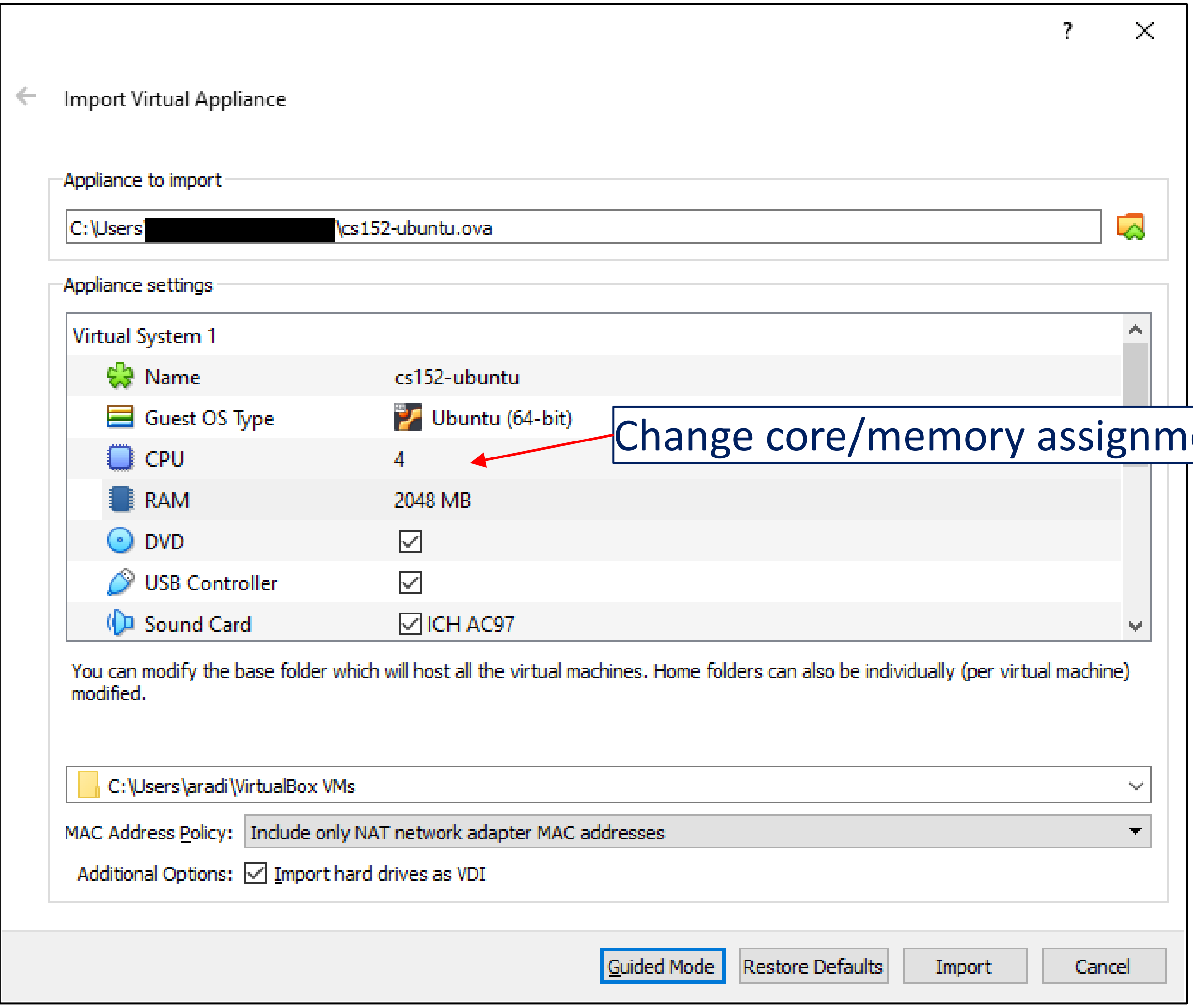

Oracle VM VirtualBox Manager
File
Machine
Help
Preferences... Ctrl+G
Import Appliance... Ctrl+I
Export Appliance... Ctrl+E
New Cloud VM...
Virtual Media Manager... Ctrl+D
Host Network Manager... Ctrl+H
Network Operations Manager...
Check for Updates...
Reset All Warnings
Exit Ctrl+Q
Export
New
Add



VirtualBox currently supports importing appliances saved in the Open Virtualization Format (OVF). To continue, select the file to import below.

If core count/memory allowance needs changing

Getting started



Import Virtual Appliance

Appliance to import

C:\Users[redacted]\cs152-ubuntu.ova

Appliance settings

Virtual System 1

Name	cs152-ubuntu
Guest OS Type	Ubuntu (64-bit)
CPU	4
RAM	2048 MB
DVD	☑
USB Controller	☑
Sound Card	☑ ICH AC97

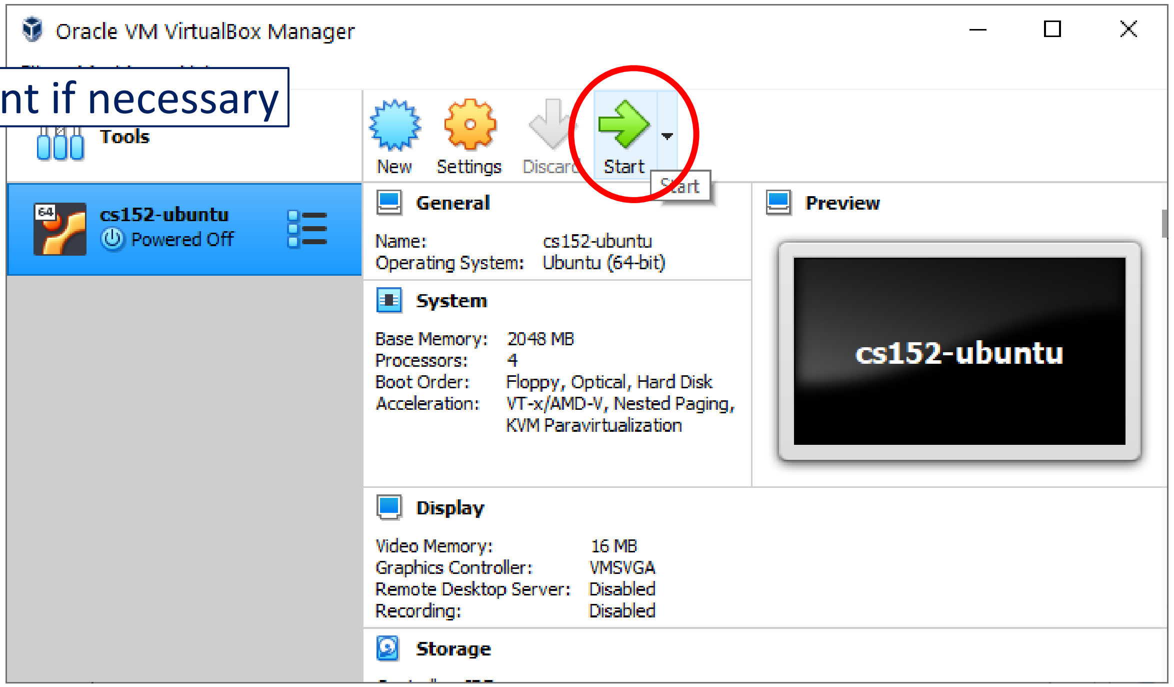
You can modify the base folder which will host all the virtual machines. Home folders can also be individually (per virtual machine) modified.

C:\Users\aradi\VirtualBox VMs

MAC Address Policy: Include only NAT network adapter MAC addresses

Additional Options: ☑ Import hard drives as VDI

Change core/memory assignment if necessary



Oracle VM VirtualBox Manager

cs152-ubuntu
Powered Off

General

Name: cs152-ubuntu
Operating System: Ubuntu (64-bit)

System

Base Memory: 2048 MB
Processors: 4
Boot Order: Floppy, Optical, Hard Disk
Acceleration: VT-x/AMD-V, Nested Paging, KVM Paravirtualization

Preview

cs152-ubuntu

Display

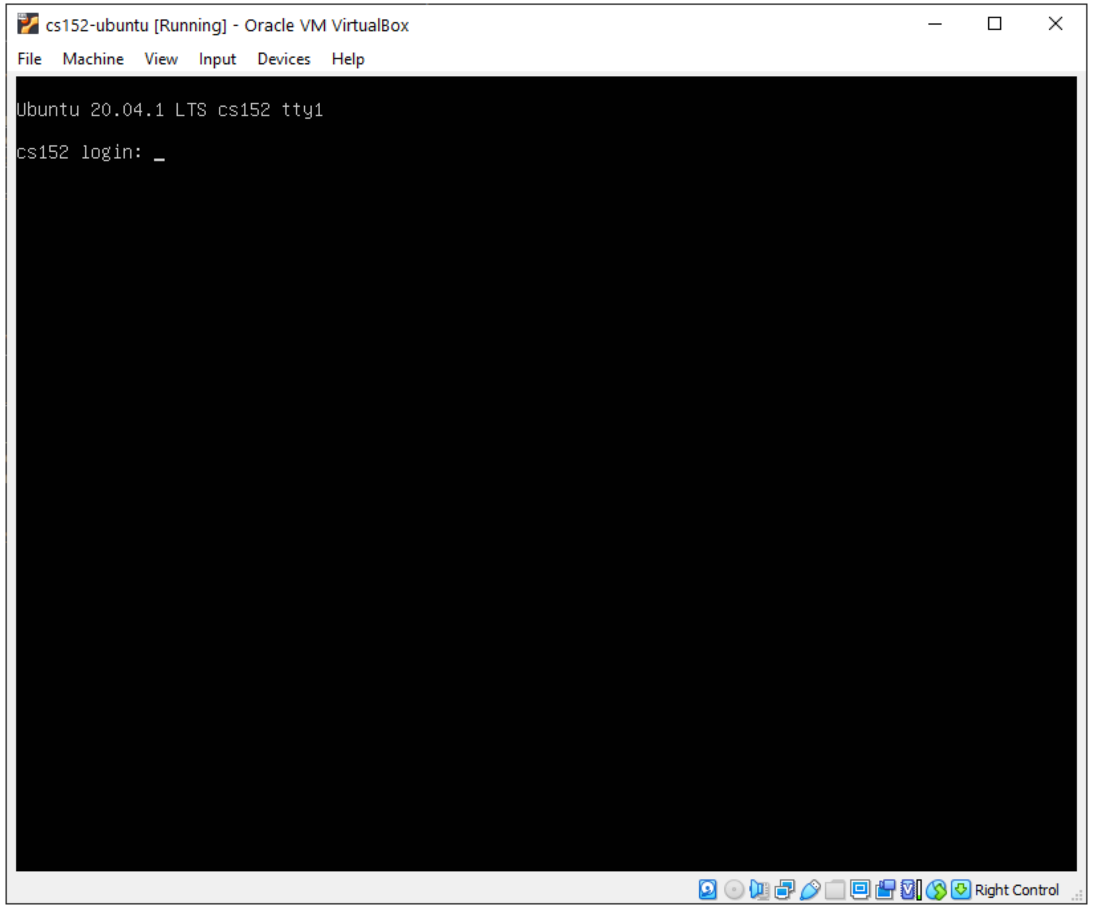
Video Memory: 16 MB
Graphics Controller: VMSVGA
Remote Desktop Server: Disabled
Recording: Disabled

Storage

Getting started

- You can work in the VM window, OR
- Connect to it via a terminal
  - Putty, MobaXterm, OpenSSH, etc
- The VM forwards its
  - port 22 (ssh) to
  - 3022
  - Connect to it by ssh cs152@127.0.0.1:3022
- Login: cs152/cs152
- Run ./clone-ulx3s.sh

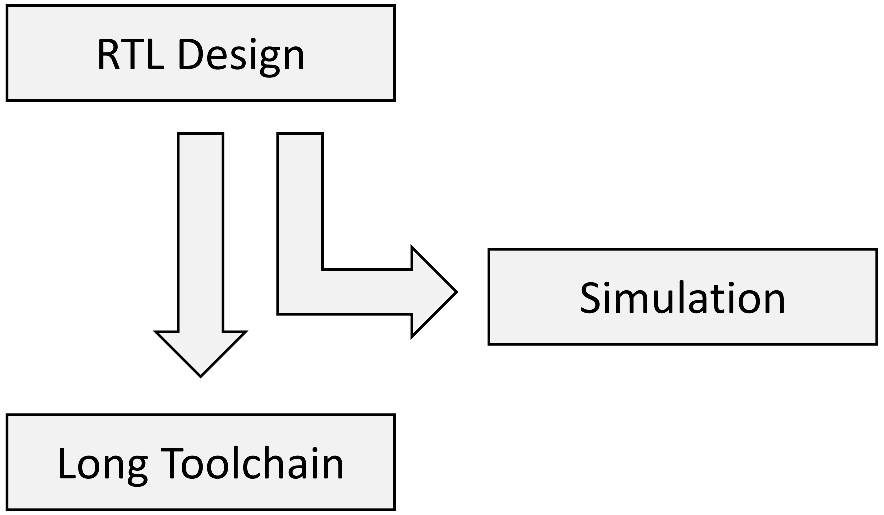
Check it out!

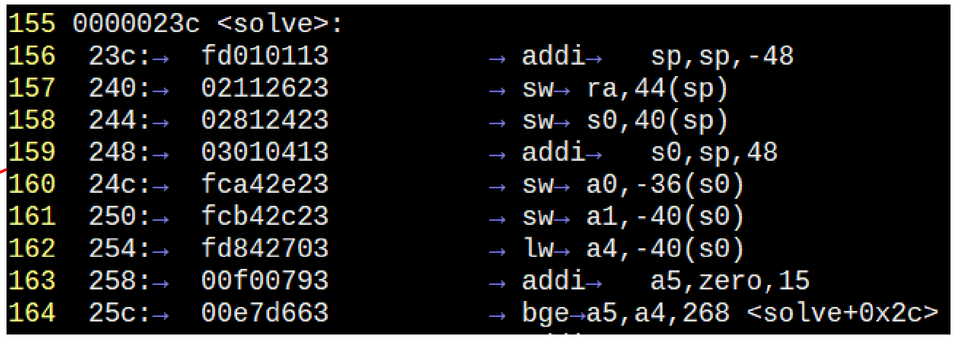

cs152-ubuntu [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Right Control

```
Ubuntu 20.04.1 LTS cs152 tty1

cs152 login: _
```

Trying simulation

- ❑ cs152-rv32i-bsv/projects/rv32i/
- ❑ Compiling and running the simulation
  - “make bsim” – Stands for “bluesim”
  - “make runsim” creates two files
    - system.log : log of processor operation
    - output.log : log of software output
- ❑ Default benchmark: Sudoku solver
  - Source: sw/minisudoku.c
  - Resulting assembly: sw/minisudoku.dump
  - Binary for processor: sw/minisudoku.bin

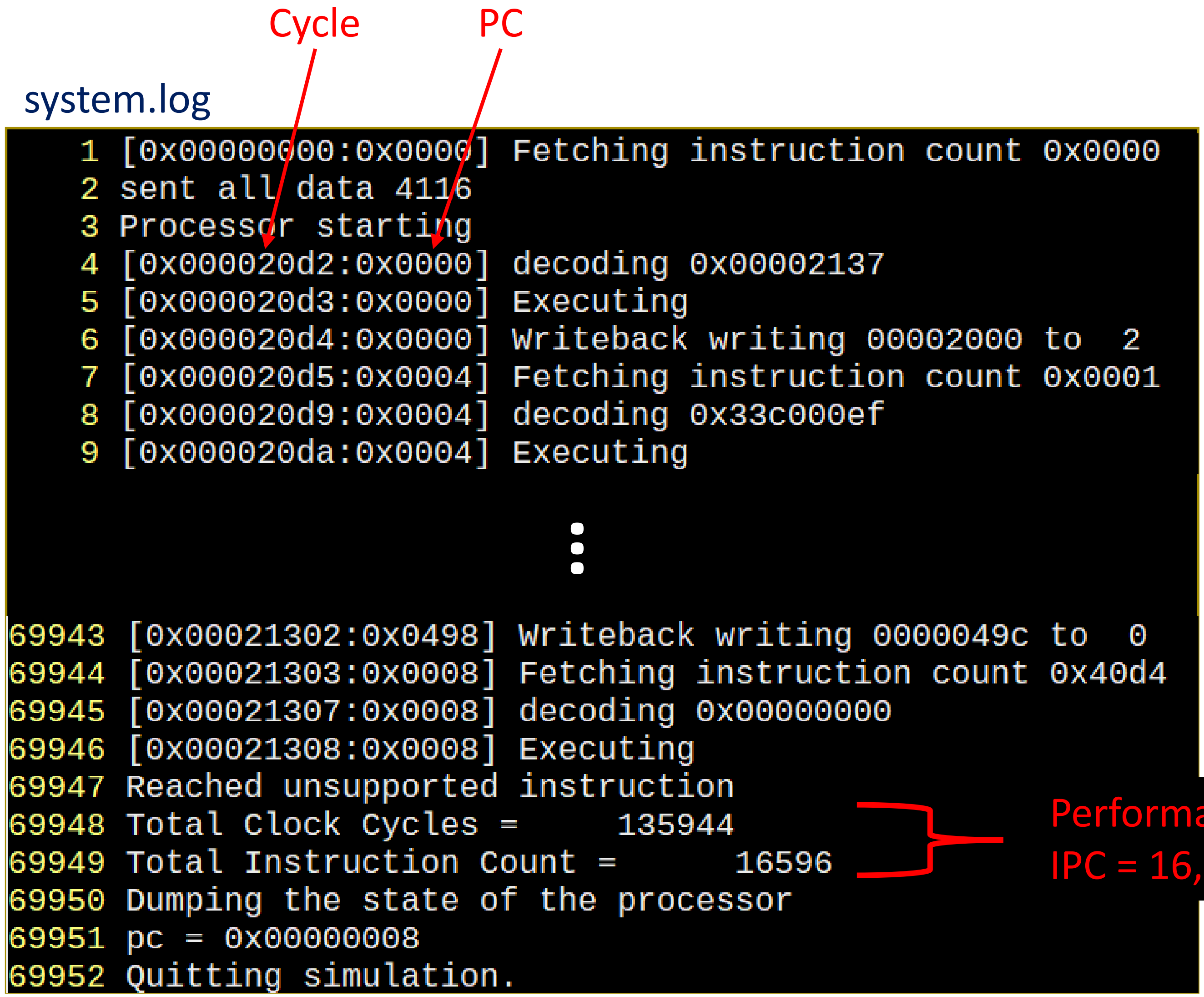




```
0000023c <solve>:
 23c:→  fd010113           → addi→   sp,sp,-48
 240:→  02112623           → sw→ ra,44(sp)
 244:→  02812423           → sw→ s0,40(sp)
 248:→  03010413           → addi→   s0,sp,48
 24c:→  fca42e23           → sw→ a0,-36(s0)
 250:→  fcb42c23           → sw→ a1,-40(s0)
 254:→  fd842703           → lw→ a4,-40(s0)
 258:→  00f00793           → addi→   a5,zero,15
 25c:→  00e7d663           → bge→a5,a4,268 <solve+0x2c>
```

Example simulation execution

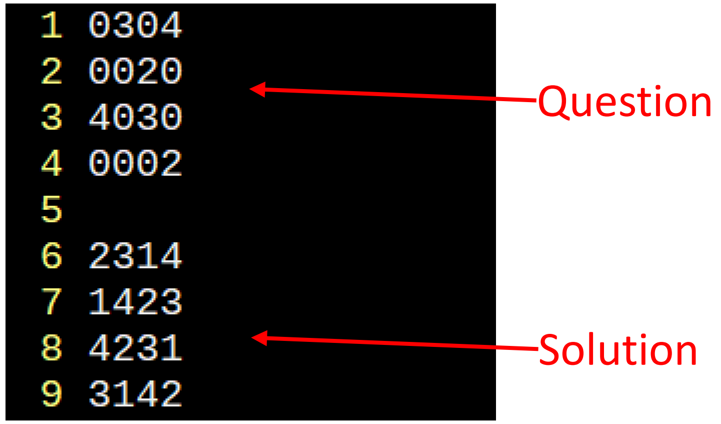
system.log



```
[0x00000000:0x0000] Fetching instruction count 0x0000
sent all data 4116
Processor starting
[0x000020d2:0x0000] decoding 0x00002137
[0x000020d3:0x0000] Executing
[0x000020d4:0x0000] Writeback writing 00002000 to  2
[0x000020d5:0x0004] Fetching instruction count 0x0001
[0x000020d9:0x0004] decoding 0x33c000ef
[0x000020da:0x0004] Executing

                          ⋮

[0x00021302:0x0498] Writeback writing 0000049c to  0
[0x00021303:0x0008] Fetching instruction count 0x40d4
[0x00021307:0x0008] decoding 0x00000000
[0x00021308:0x0008] Executing
Reached unsupported instruction
Total Clock Cycles =      135944
Total Instruction Count =      16596
Dumping the state of the processor
pc = 0x00000008
Quitting simulation.
```

Performance numbers!
IPC = 16,596 / 135,944 ~= 0.122

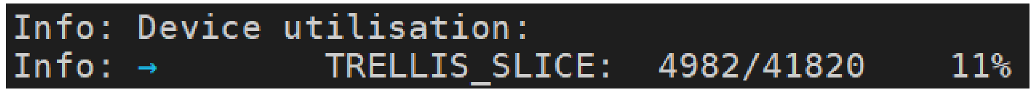
output.log



```
0304
0020
4030
0002

2314
1423
4231
3142
```

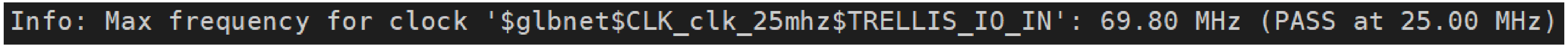
Trying synthesis

- ❑ Synthesis to hardware
  - “make | tee build.log”
  - Log file is long!
- ❑ Example log files from synthesis:
  - Look for “Device utilisation” [sic]:



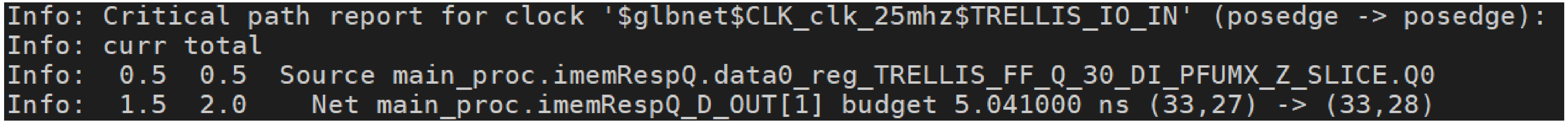
```
Info: Device utilisation:
Info: →          TRELLIS_SLICE:  4982/41820    11%
```

  - Look for “Max frequency” :



```
Info: Max frequency for clock '$glbnet$CLK_clk_25mhz$TRELLIS_IO_IN': 69.80 MHz (PASS at 25.00 MHz)
```

  - Look for “Critical path report for clock”:



```
Info: Critical path report for clock '$glbnet$CLK_clk_25mhz$TRELLIS_IO_IN' (posedge -> posedge):
Info: curr total
Info:  0.5  0.5  Source main_proc.imemRespQ.data0_reg_TRELLIS_FF_Q_30_DI_PFUMX_Z_SLICE.Q0
Info:  1.5  2.0    Net main_proc.imemRespQ_D_OUT[1] budget 5.041000 ns (33,27) -> (33,28)
```

Measuring the performance of our processor

- From the **simulation**, we can measure the clock cycles to completion
- From **synthesis**, we can measure the clock speed
- (cycle count)/(clock frequency) = time to completion!

- In our previous example, 135,944 cycles / 69.80 MHz = 0.0019s
  - Is this good?
  - We can do MUCH better!

CS152: Computer Systems Architecture
Dive Into The Example Processor

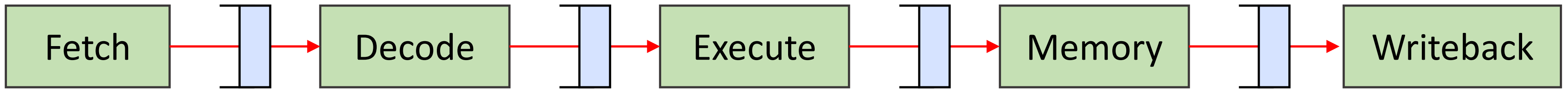
Sang-Woo Jun

Winter 2022

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

Goal of these exercises

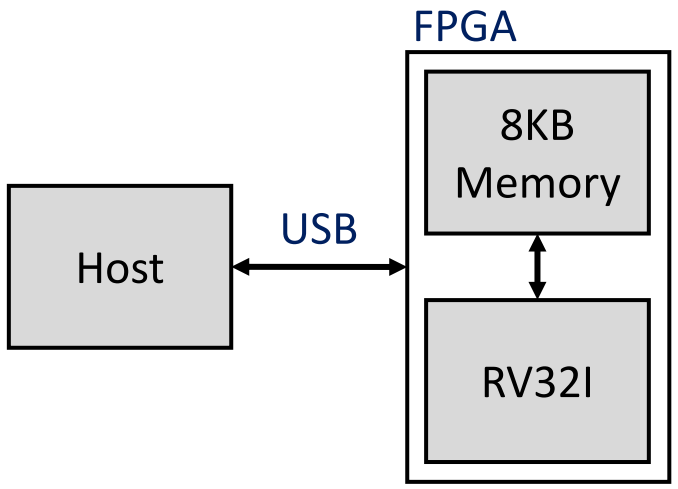
- Lots of details are lost when described at a high level
  - E.g., What information is sent between execute and memory stages?



- Experience the performance impact of modifications
  - Clock speed? Cycle count?
  - Instruction count won't change since we're working with the same software binary
  - Time = clock period * cycle count * instruction count
- I will guide you through pipelining, but not comment on performance
  - See for yourself!

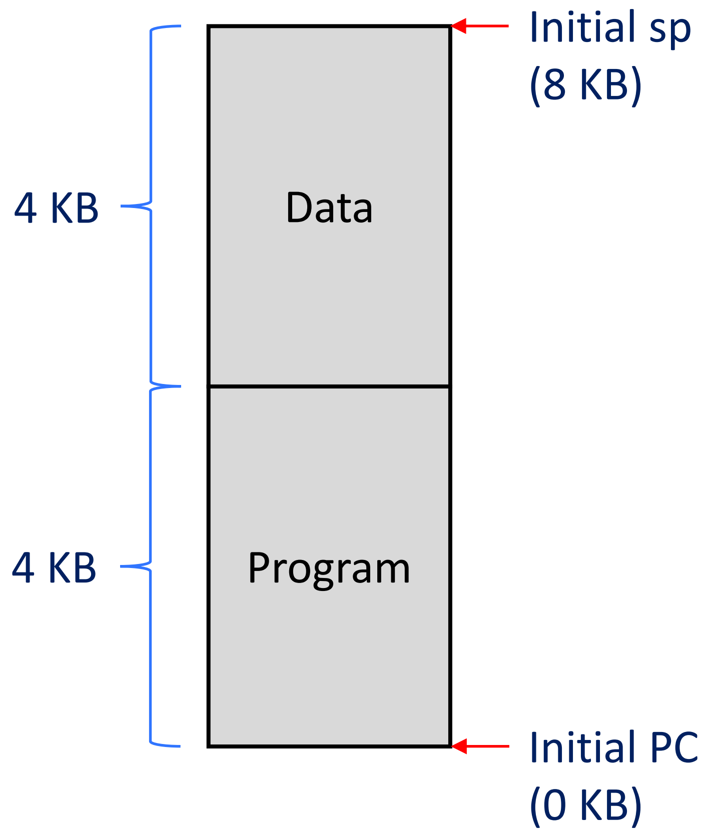
Hardware platform overview

- Lattice ECP5-85F FPGA
- Host software loads software/data over USB to FPGA
- Configured with limited on-chip memory
  - 8 KB on-chip memory
    - Arbitrary choice… Hardware can support much more
    - Enough for sudoku!



Processor memory map

- ❑ Memory space divided into program and data
  - ○ 4 KB each
- ❑ Host software loads program and data
- ❑ And then starts processor
- ❑ No writes allowed in program space
  - ○ All writes to program are MMIO’d into software
  - ○ Simply printed to screen at host



Processor code structure

- cs152-rv32i-bsv/
  - projects/
    - rv32i/
      - processor/ -- Bluespec files for processor (Pipeline, register file, etc) <- You will work here
      - sw/ -- Software benchmarks (sudoku)
      - cpp/ -- Host software
  - src/ -- Helper modules (USB communication, memory module, etc)

The big principle in hardware design

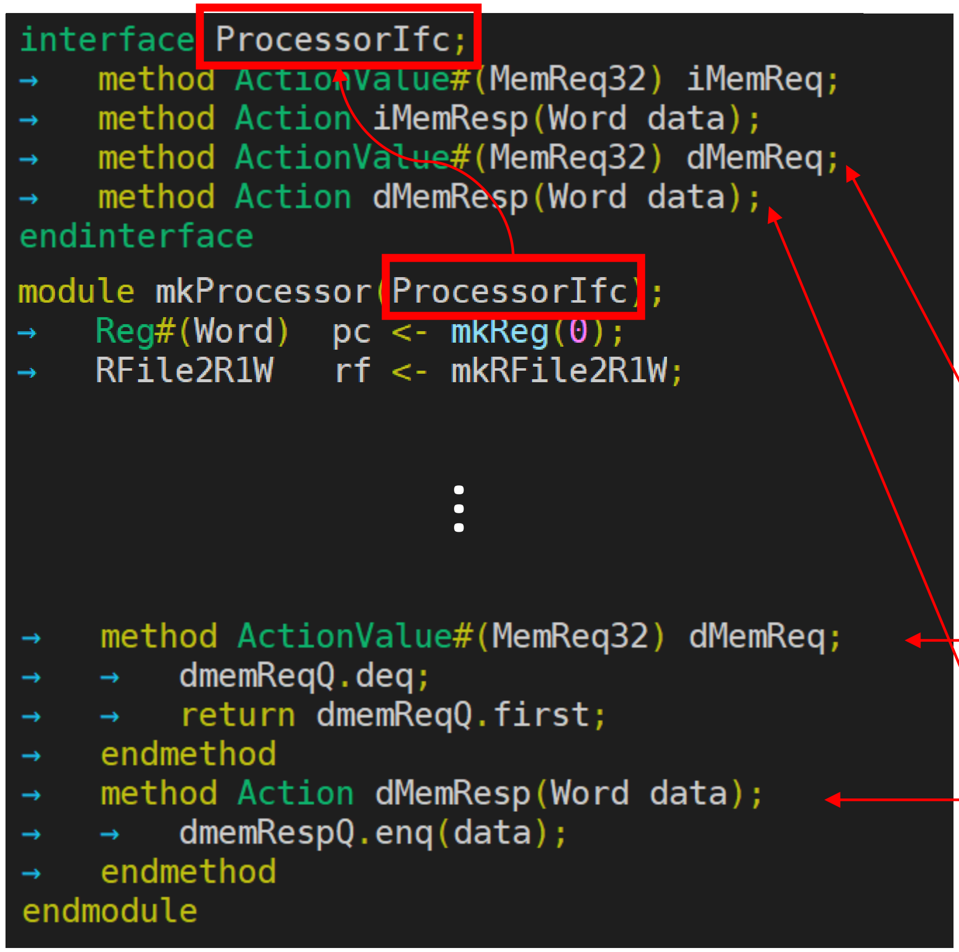
- **EVERYTHING is parallel!**

- All function calls, all rule executions, all method polls, …

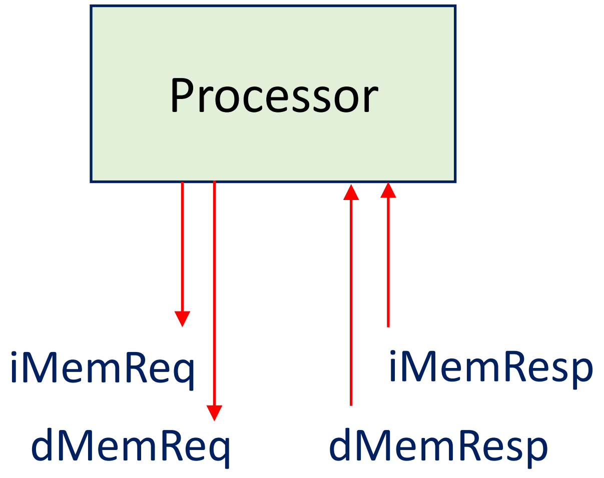
- If there are 10,000 rules (~= ‘always’ blocks),
  ideally 10,000 rules will all be executing **EVERY cycle**

Basic microarchitecture in Bluespec: The interface

Projects/rv32i/processor/Processor.bsv



```
interface ProcessorIfc;
    method ActionValue#(MemReq32) iMemReq;
    method Action iMemResp(Word data);
    method ActionValue#(MemReq32) dMemReq;
    method Action dMemResp(Word data);
endinterface
module mkProcessor(ProcessorIfc);
    Reg#(Word)  pc <- mkReg(0);
    RFile2R1W   rf <- mkRFile2R1W;

                    ⋮

    method ActionValue#(MemReq32) dMemReq;
        dmemReqQ.deq;
        return dmemReqQ.first;
    endmethod
    method Action dMemResp(Word data);
        dmemRespQ.enq(data);
    endmethod
endmodule
```



Outside environment polls this method for memory requests

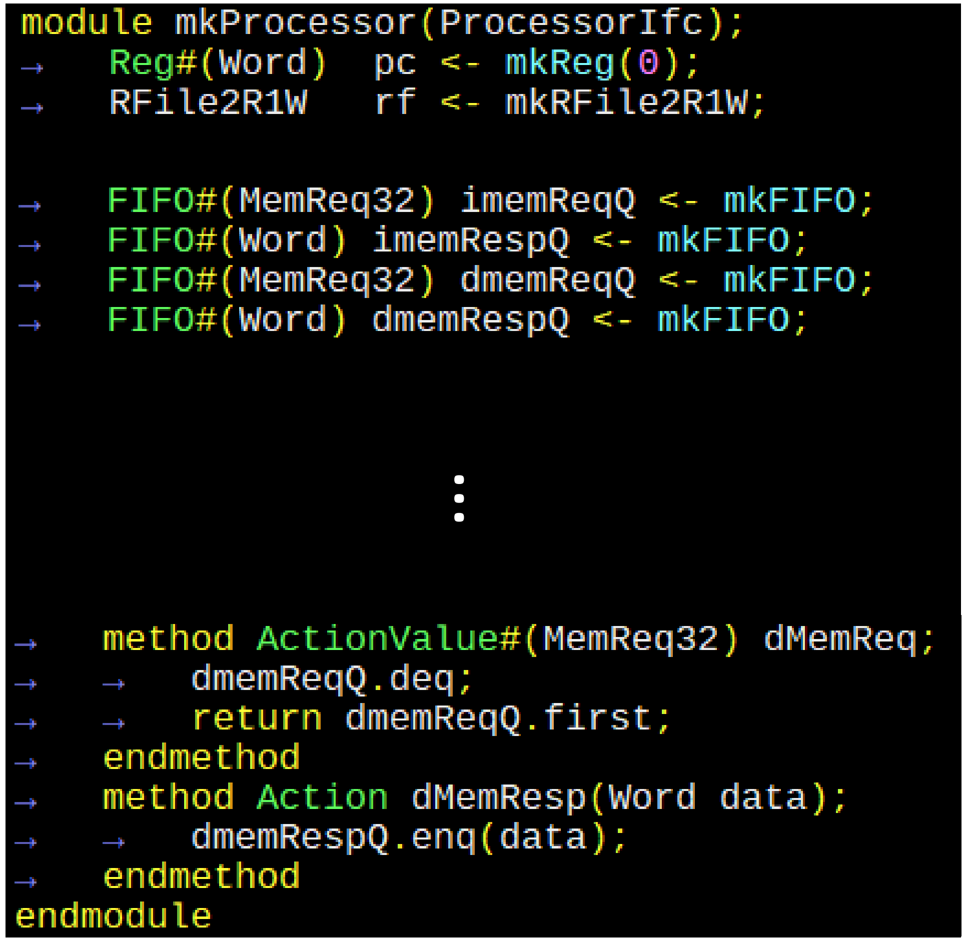
Memory responses arrive in the processor

(Processor can enqueue memory requests into dmemReqQ)

Everything outside the processor is provided

Basic microarchitecture in Bluespec: The interface

Projects/rv32i/processor/Processor.bsv



```
module mkProcessor(ProcessorIfc);
→   Reg#(Word)  pc <- mkReg(0);
→   RFile2R1W   rf <- mkRFile2R1W;

→   FIFO#(MemReq32) imemReqQ <- mkFIFO;
→   FIFO#(Word) imemRespQ <- mkFIFO;
→   FIFO#(MemReq32) dmemReqQ <- mkFIFO;
→   FIFO#(Word) dmemRespQ <- mkFIFO;

                    ⋮

→   method ActionValue#(MemReq32) dMemReq;
→   →   dmemReqQ.deq;
→   →   return dmemReqQ.first;
→   endmethod
→   method Action dMemResp(Word data);
→   →   dmemRespQ.enq(data);
→   endmethod
endmodule
```

Register of type “Word” (32 bits)

Register file

FIFOs of Memory Req types and Word types
Default size is 2

Types are defined in processor/Defines.bsv

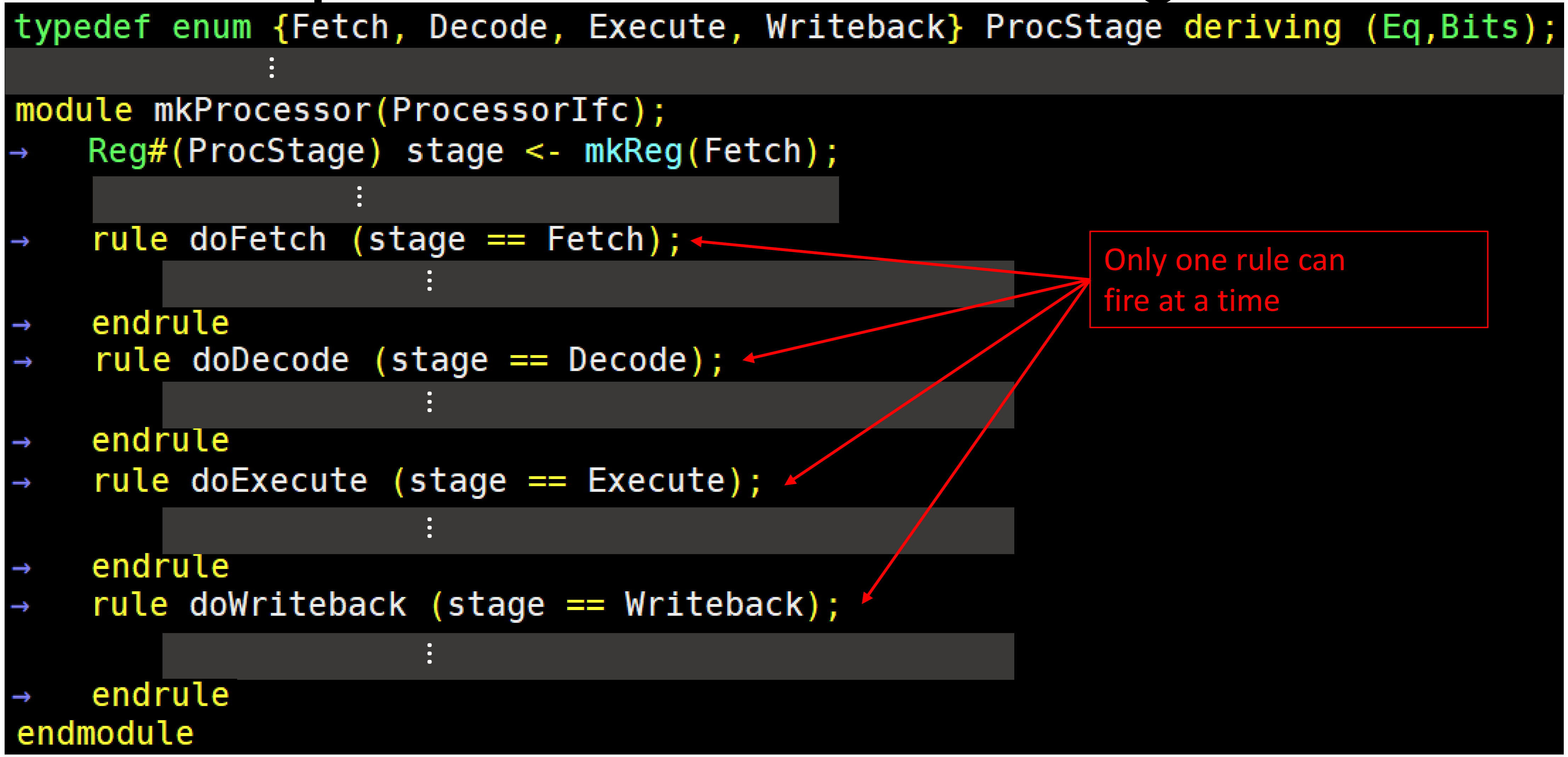
- Processor can make instruction and data memory requests via imemReqQ and dmemReqQ
- Responses will arrive via imemRespQ and dmemRespQ

Basic microarchitecture in Bluespec: The stages

- A 4-stage implementation is provided
  - Execute and memory merged into Execute for simplicity
    - Good idea?
  - Expressed via four **‘rules’**
    - doFetch
    - doDecode
    - doExecute
    - doWriteback

- Not yet pipelined: Goal of the labs!

Basic microarchitecture in Bluespec: Rules express combinational logic

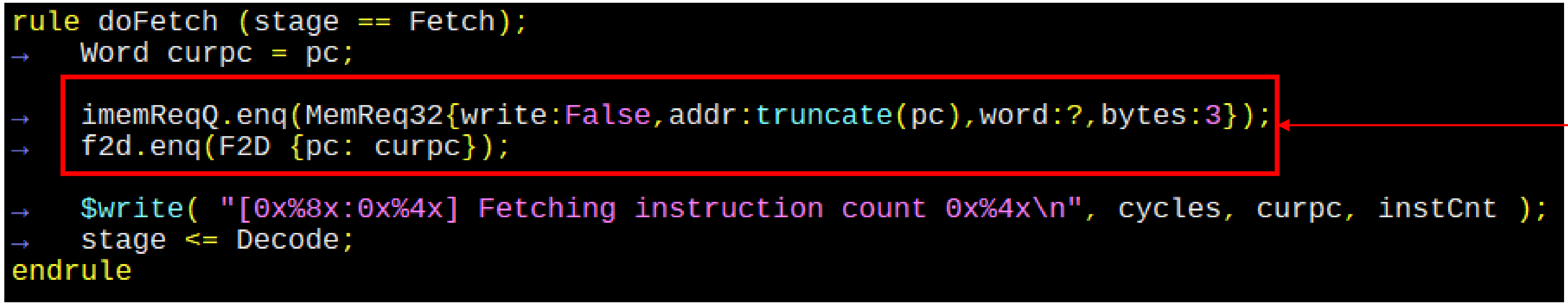


```
typedef enum {Fetch, Decode, Execute, Writeback} ProcStage deriving (Eq,Bits);
                  ⋮
module mkProcessor(ProcessorIfc);
→    Reg#(ProcStage) stage <- mkReg(Fetch);
                  ⋮
→    rule doFetch (stage == Fetch);
                  ⋮
→    endrule
→    rule doDecode (stage == Decode);
                  ⋮
→    endrule
→    rule doExecute (stage == Execute);
                  ⋮
→    endrule
→    rule doWriteback (stage == Writeback);
                  ⋮
→    endrule
endmodule
```

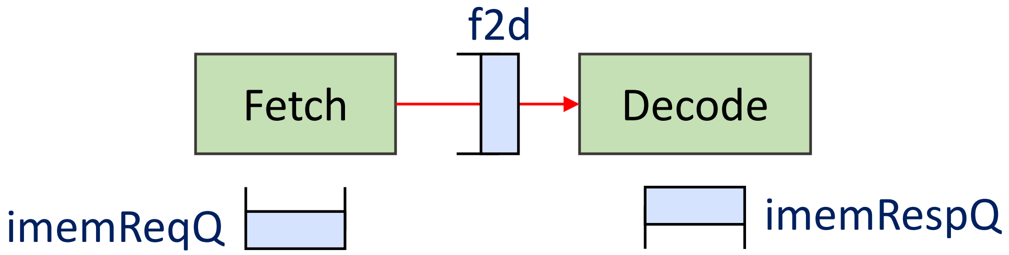
The fetch stage

IMPORTANT!
Rules express combinational circuits
Meaning there is no ordering between expressions!
(Unless there is dependency)

- ❑ Sends memory req via imemReqQ
- ❑ Enqs into pipeline FIFO f2d
  - ○ Same naming convention between other stages (f2d, d2e, e2m)

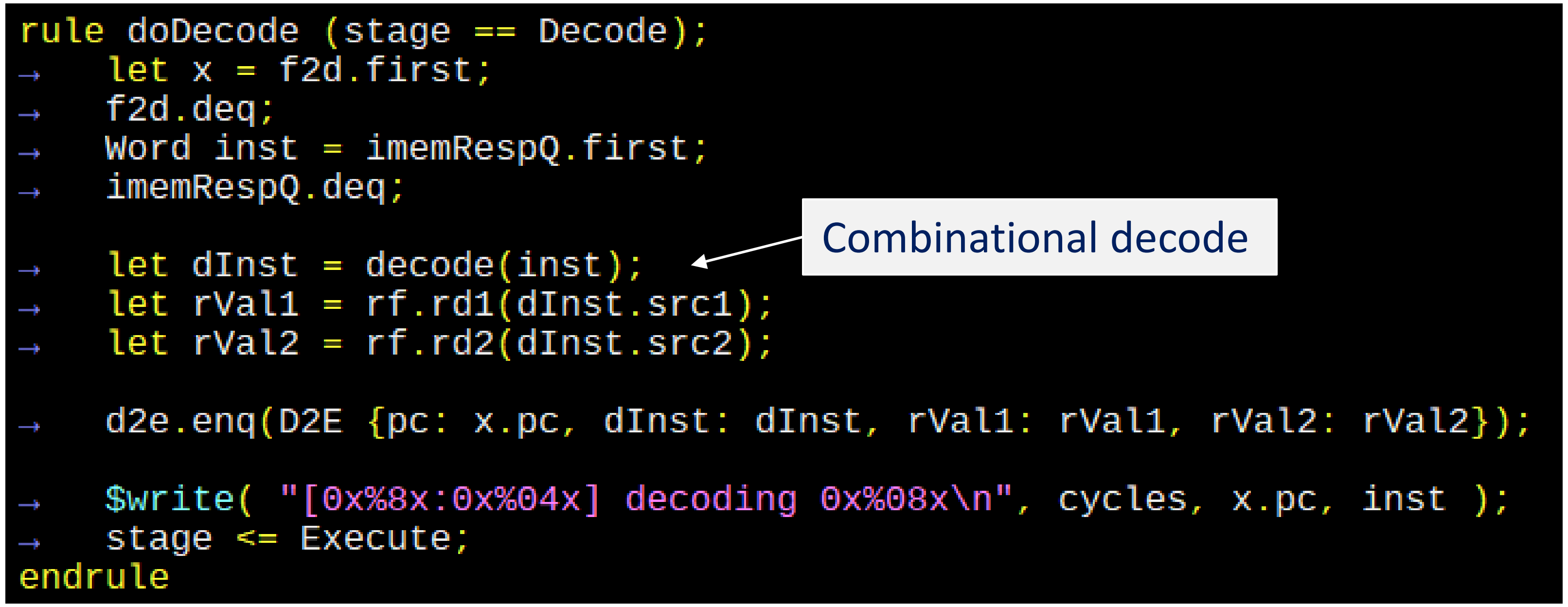


```
rule doFetch (stage == Fetch);
→   Word curpc = pc;

→   imemReqQ.enq(MemReq32{write:False,addr:truncate(pc),word:?,bytes:3});
→   f2d.enq(F2D {pc: curpc});

→   $write( "[0x%8x:0x%4x] Fetching instruction count 0x%4x\n", cycles, curpc, instCnt );
→   stage <= Decode;
endrule
```



The decode stage

- “decode” function defined in processor/Decode.bsv
  - Extracts bit-encoded information and expands it into an easy-to-use structure

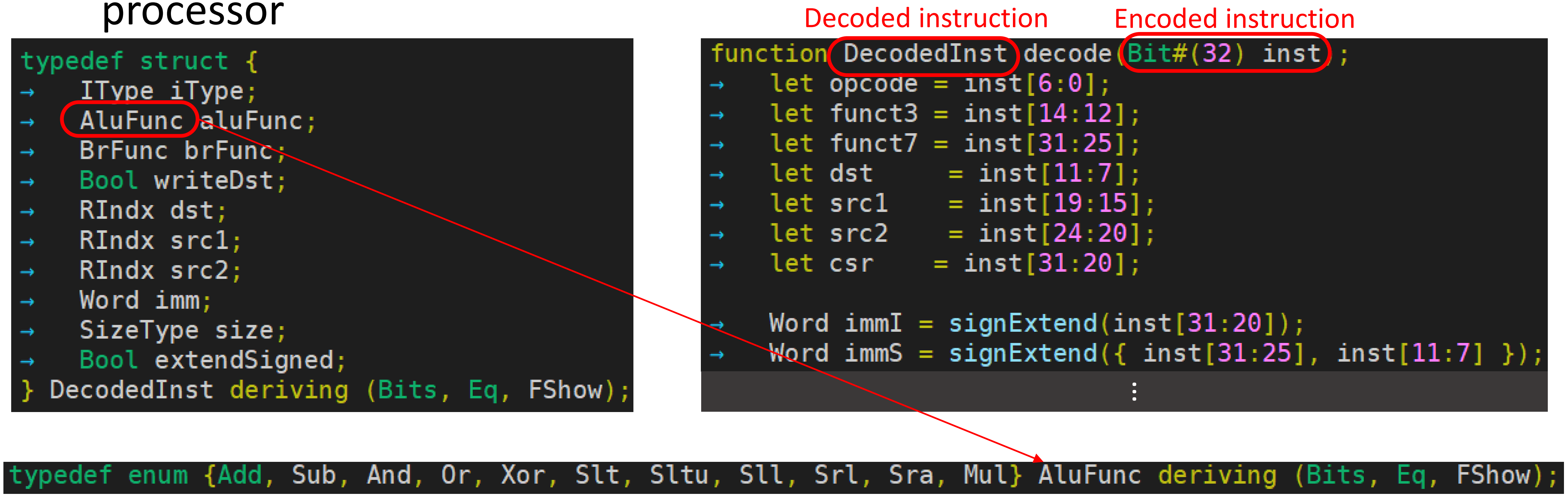


```
rule doDecode (stage == Decode);
    let x = f2d.first;
    f2d.deq;
    Word inst = imemRespQ.first;
    imemRespQ.deq;

    let dInst = decode(inst);
    let rVal1 = rf.rd1(dInst.src1);
    let rVal2 = rf.rd2(dInst.src2);

    d2e.enq(D2E {pc: x.pc, dInst: dInst, rVal1: rVal1, rVal2: rVal2});

    $write( "[0x%8x:0x%04x] decoding 0x%08x\n", cycles, x.pc, inst );
    stage <= Execute;
endrule
```

- Let’s look at code! (Decode.bsv)

The decode function

- Analyzes the 32-bit encoded instruction
- Returns a decoded instruction that is easier to use by the rest of the processor



Decoded instruction

Encoded instruction

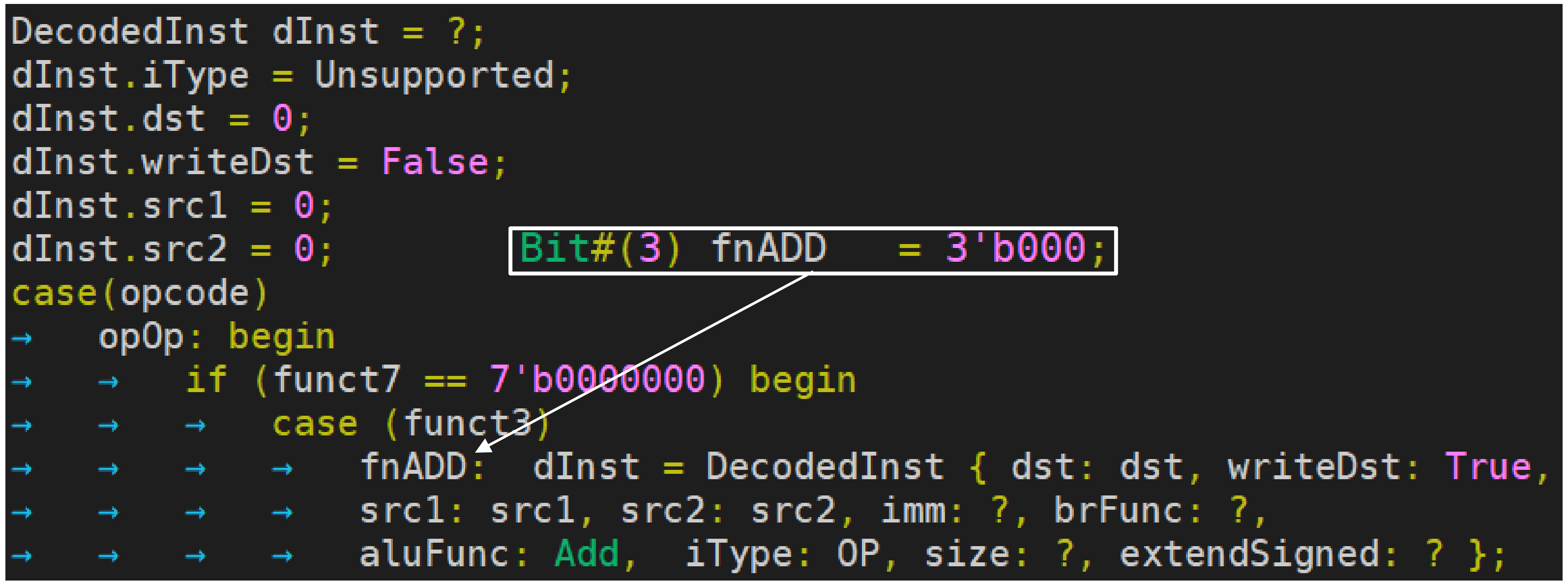
```
typedef struct {
→   IType iType;
→   AluFunc aluFunc;
→   BrFunc brFunc;
→   Bool writeDst;
→   RIndx dst;
→   RIndx src1;
→   RIndx src2;
→   Word imm;
→   SizeType size;
→   Bool extendSigned;
} DecodedInst deriving (Bits, Eq, FShow);
```

```
function DecodedInst decode(Bit#(32) inst);
→   let opcode = inst[6:0];
→   let funct3 = inst[14:12];
→   let funct7 = inst[31:25];
→   let dst     = inst[11:7];
→   let src1    = inst[19:15];
→   let src2    = inst[24:20];
→   let csr    = inst[31:20];

→   Word immI = signExtend(inst[31:20]);
→   Word immS = signExtend({ inst[31:25], inst[11:7] });
                              ⋮
```

```
typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra, Mul} AluFunc deriving (Bits, Eq, FShow);
```

The decode function – Example

- Add instruction: funct7 == 0 && funct3 == 0
  - Dst, src1, src2 exists, Instruction type is “OP” (register-register operation)
  - aluFunc is Add
  - No imm, size
  - Not branch instruction (BEQ, BNE, etc)

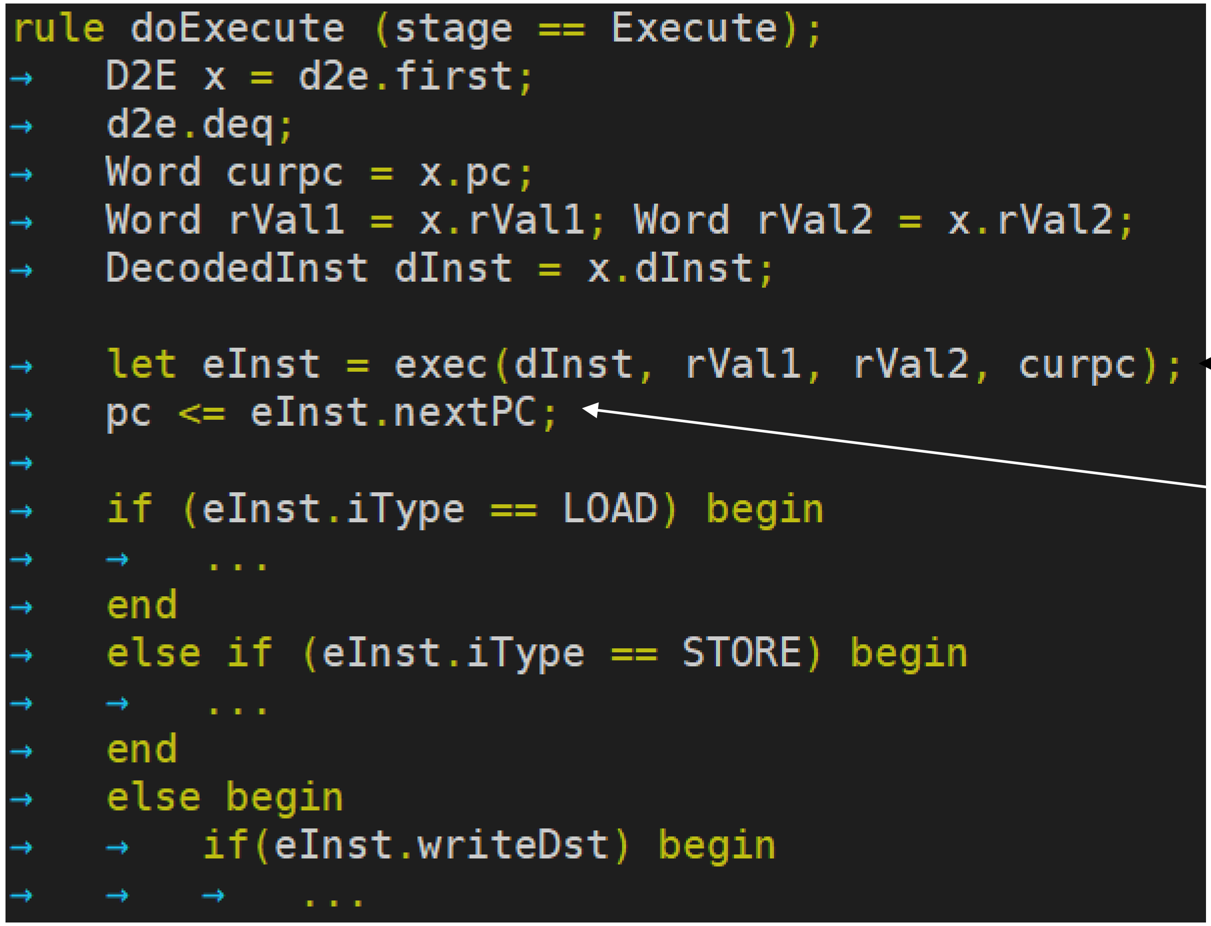


```
DecodedInst dInst = ?;
dInst.iType = Unsupported;
dInst.dst = 0;
dInst.writeDst = False;
dInst.src1 = 0;
dInst.src2 = 0;          Bit#(3) fnADD    = 3'b000;
case(opcode)
→   opOp: begin
→   →   if (funct7 == 7'b0000000) begin
→   →   →   case (funct3)
→   →   →   →   fnADD:  dInst = DecodedInst { dst: dst, writeDst: True,
→   →   →   →   src1: src1, src2: src2, imm: ?, brFunc: ?,
→   →   →   →   aluFunc: Add,  iType: OP, size: ?, extendSigned: ? };
```

R-Type encoding	funct7	rs2	rs1	funct3	rd	opcode
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits
e.g., add x9,x20,x21	0	21	20	0	9	51

The execute stage

- “exec” implements ALU operations (in processor/Execute.bsv)



```
rule doExecute (stage == Execute);
→   D2E x = d2e.first;
→   d2e.deq;
→   Word curpc = x.pc;
→   Word rVal1 = x.rVal1; Word rVal2 = x.rVal2;
→   DecodedInst dInst = x.dInst;

→   let eInst = exec(dInst, rVal1, rVal2, curpc);
→   pc <= eInst.nextPC;
→
→   if (eInst.iType == LOAD) begin
→   →   ...
→   end
→   else if (eInst.iType == STORE) begin
→   →   ...
→   end
→   else begin
→   →   if(eInst.writeDst) begin
→   →   →   ...
```

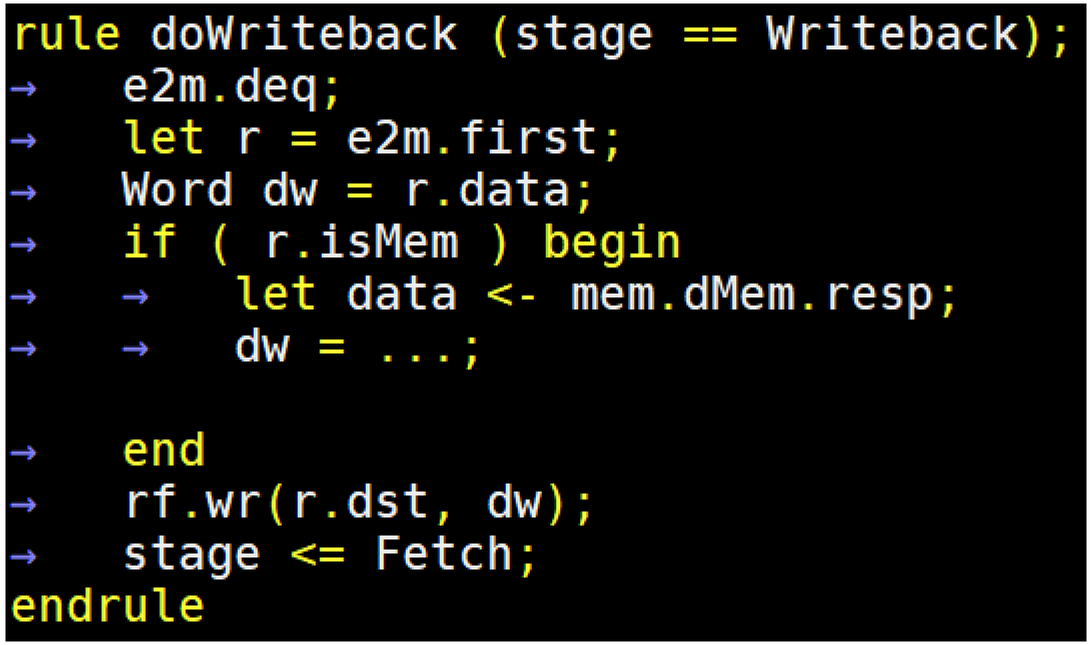
Bluespec functions are combinational circuits (No state changes)

non-pipelined version always sets pc for fetch

Take a look at processor/Execute.bsv!

The writeback stage

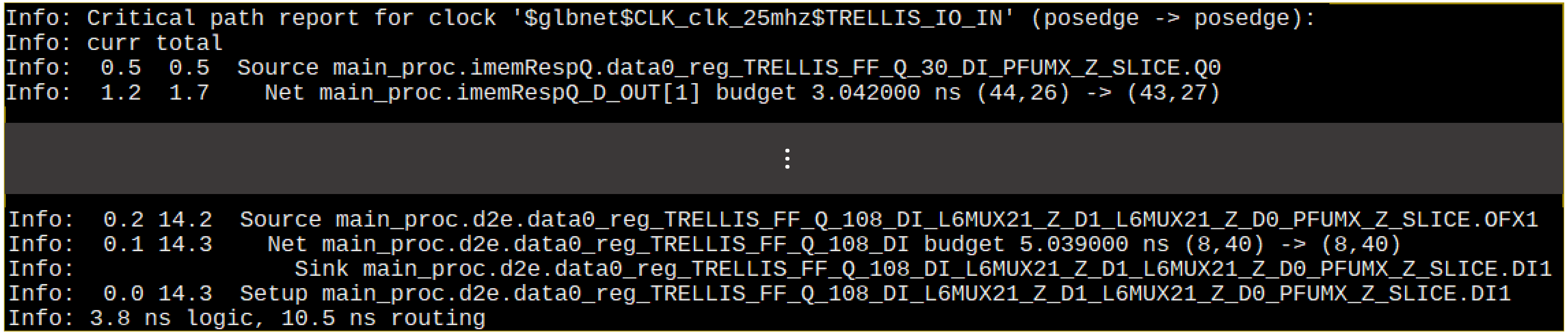
- Straightforward enough!
  - Let’s look at code! And notice handling of signed/unsigned numbers



```
rule doWriteback (stage == Writeback);
    e2m.deq;
    let r = e2m.first;
    Word dw = r.data;
    if ( r.isMem ) begin
        let data <- mem.dMem.resp;
        dw = ...;

    end
    rf.wr(r.dst, dw);
    stage <= Fetch;
endrule
```

Aside: Looking back at the critical path

- Which stage is the critical path?
  - Look at the synthesis log!
- Was it a good idea to merge execute and memory?

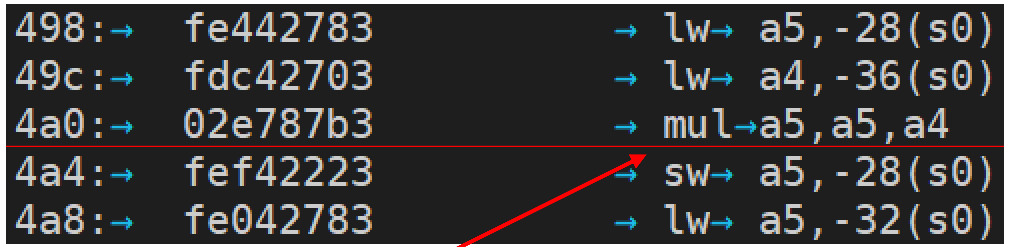


```
Info: Critical path report for clock '$glbnet$CLK_clk_25mhz$TRELLIS_IO_IN' (posedge -> posedge):
Info: curr total
Info:  0.5  0.5  Source main_proc.imemRespQ.data0_reg_TRELLIS_FF_Q_30_DI_PFUMX_Z_SLICE.Q0
Info:  1.2  1.7    Net main_proc.imemRespQ_D_OUT[1] budget 3.042000 ns (44,26) -> (43,27)
⋮
Info:  0.2 14.2  Source main_proc.d2e.data0_reg_TRELLIS_FF_Q_108_DI_L6MUX21_Z_D1_L6MUX21_Z_D0_PFUMX_Z_SLICE.OFX1
Info:  0.1 14.3    Net main_proc.d2e.data0_reg_TRELLIS_FF_Q_108_DI budget 5.039000 ns (8,40) -> (8,40)
Info:                Sink main_proc.d2e.data0_reg_TRELLIS_FF_Q_108_DI_L6MUX21_Z_D1_L6MUX21_Z_D0_PFUMX_Z_SLICE.DI1
Info:  0.0 14.3  Setup main_proc.d2e.data0_reg_TRELLIS_FF_Q_108_DI_L6MUX21_Z_D1_L6MUX21_Z_D0_PFUMX_Z_SLICE.DI1
Info: 3.8 ns logic, 10.5 ns routing
```

Looking at sample execution

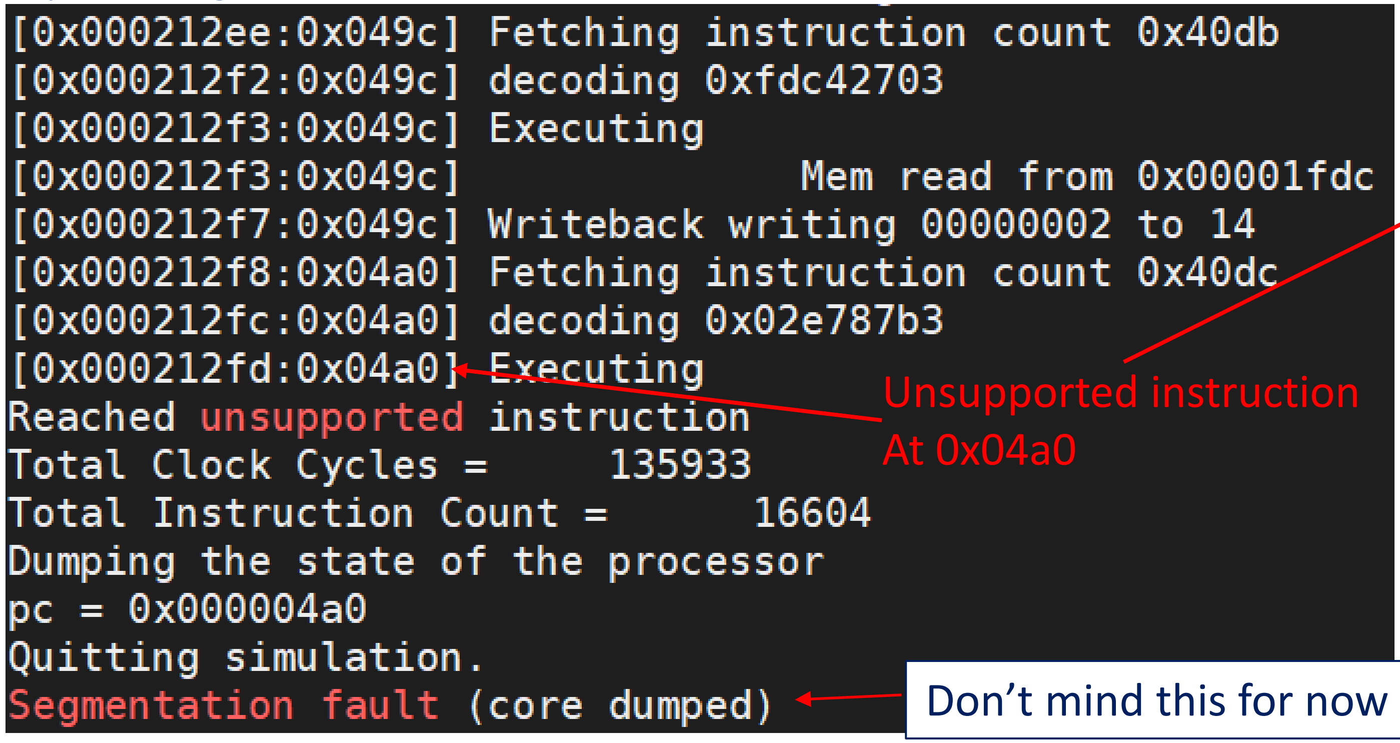
- ❑ Try running "make runsim"
- ❑ "Mul" not part of rv32i!

sw/minisudoku.dump



```
498:→   fe442783          → lw→  a5,-28(s0)
49c:→   fdc42703          → lw→  a4,-36(s0)
4a0:→   02e787b3          → mul→a5,a5,a4
4a4:→   fef42223          → sw→  a5,-28(s0)
4a8:→   fe042783          → lw→  a5,-32(s0)
```

system.log

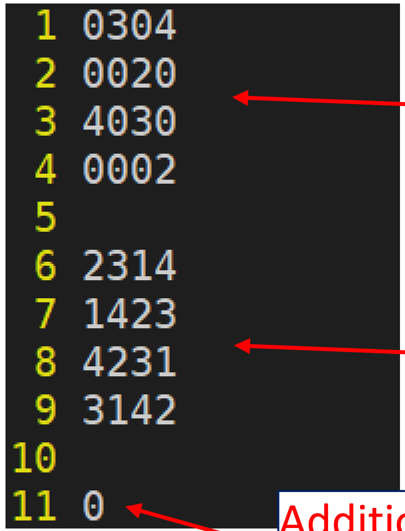


```
[0x000212ee:0x049c] Fetching instruction count 0x40db
[0x000212f2:0x049c] decoding 0xfdc42703
[0x000212f3:0x049c] Executing
[0x000212f3:0x049c]             Mem read from 0x00001fdc
[0x000212f7:0x049c] Writeback writing 00000002 to 14
[0x000212f8:0x04a0] Fetching instruction count 0x40dc
[0x000212fc:0x04a0] decoding 0x02e787b3
[0x000212fd:0x04a0] Executing
Reached unsupported instruction
Total Clock Cycles =     135933
Total Instruction Count =       16604
Dumping the state of the processor
pc = 0x000004a0
Quitting simulation.
Segmentation fault (core dumped)
```

Unsupported instruction
At 0x04a0

Don't mind this for now

output.log



```
 1 0304
 2 0020
 3 4030
 4 0002
 5
 6 2314
 7 1423
 8 4231
 9 3142
10
11 0
```

Question

Solution

Additional output
With Mul implemented

First task for lab 2: Implement “Mul”

- Hint: Must change “Decode.bsv” and “Execute.bsv”
- Decode.bsv:
  - Opcode of Mul is “opOp” (Like “add” and others)
  - Funct7 is 7'b0000001 (7 bit value of 1)
  - Funct3 is 3’b000 (3 bit value of 0), already provided with name “fnMUL”
  - “Mul” is already added to enum AluFunc
  - Hint: Decoded results are very similar to, say, Add
- Execute.bsv
  - Mul should have an “OP” iType, which is an ALU operation
  - “function Word alu” in Execute should be changed to perform Mul

CS152: Computer Systems Architecture
Pipelining The Processor

Sang-Woo Jun

Winter 2022

UCI

Large amount of material adapted from MIT 6.004, “Computation Structures”,
Morgan Kaufmann “Computer Organization and Design: The Hardware/Software Interface: RISC-V Edition”,
and CS 152 Slides by Isaac Scherson

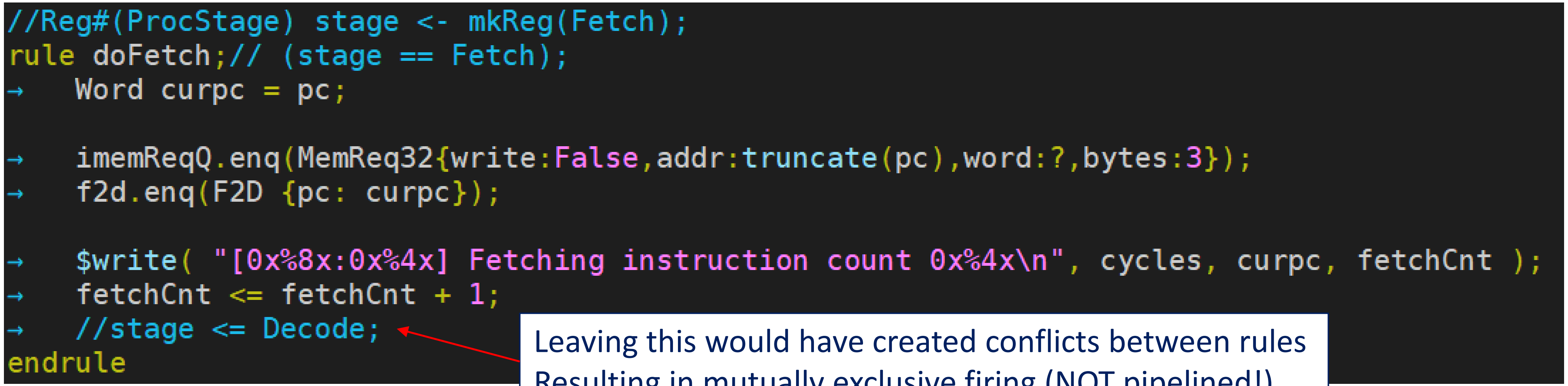
Let’s start pipelining

- Start with handling branch hazards
  - Data hazards produce wrong results,
  - but without handling branch hazards we cannot pipeline things at all
    - Which address should Fetch read?

- Things to solve:
  1. Branch hazard
  2. Load-Use hazard
  3. Read-After-Write hazard

Step 1: Simply remove guards

- ❑ Remove register “stage”, and all references to it

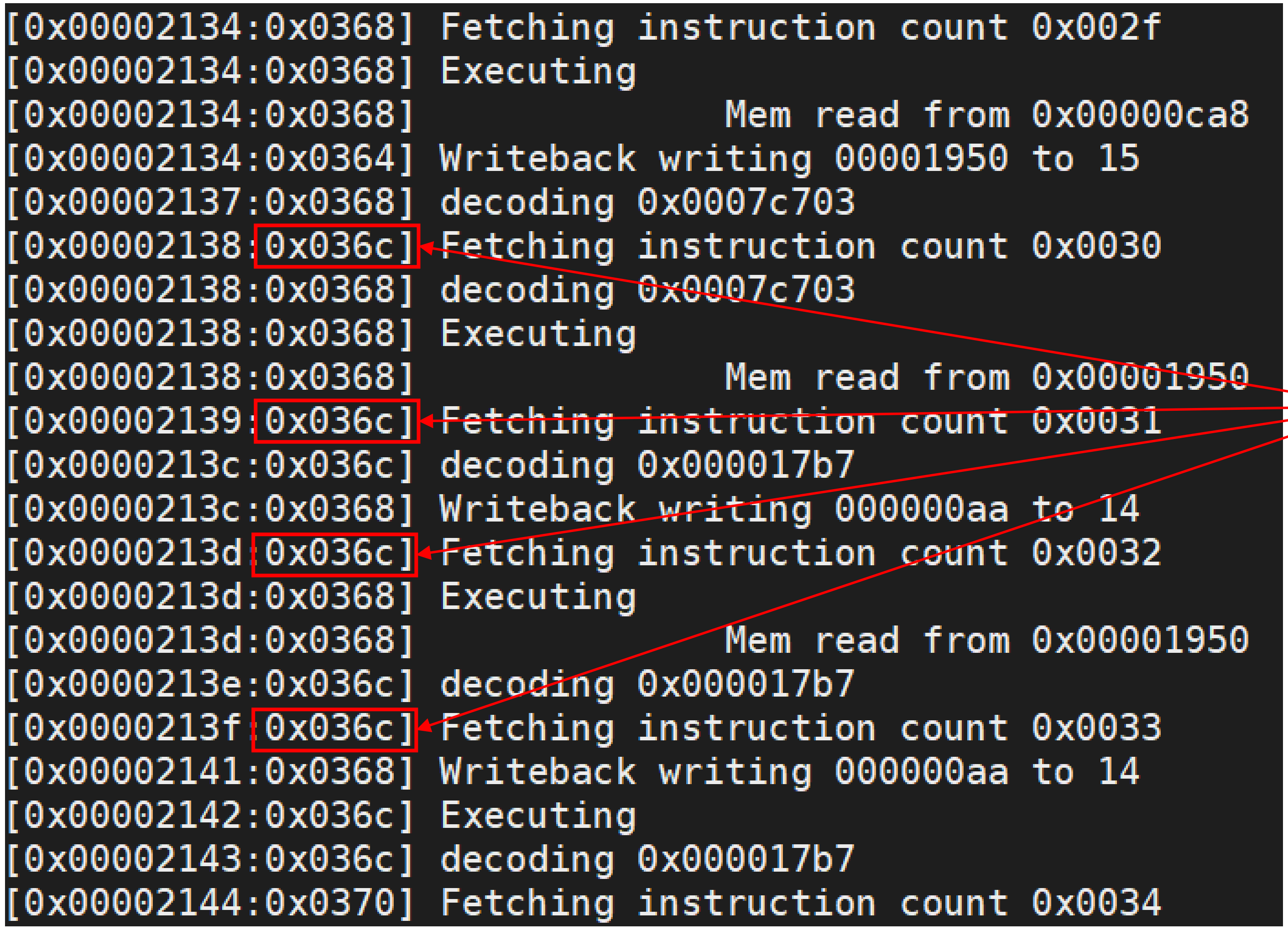


```
//Reg#(ProcStage) stage <- mkReg(Fetch);
rule doFetch;// (stage == Fetch);
    Word curpc = pc;

    imemReqQ.enq(MemReq32{write:False,addr:truncate(pc),word:?,bytes:3});
    f2d.enq(F2D {pc: curpc});

    $write( "[0x%8x:0x%4x] Fetching instruction count 0x%4x\n", cycles, curpc, fetchCnt );
    fetchCnt <= fetchCnt + 1;
    //stage <= Decode;
endrule
```

Leaving this would have created conflicts between rules
Resulting in mutually exclusive firing (NOT pipelined!)

Did that work?

system.log



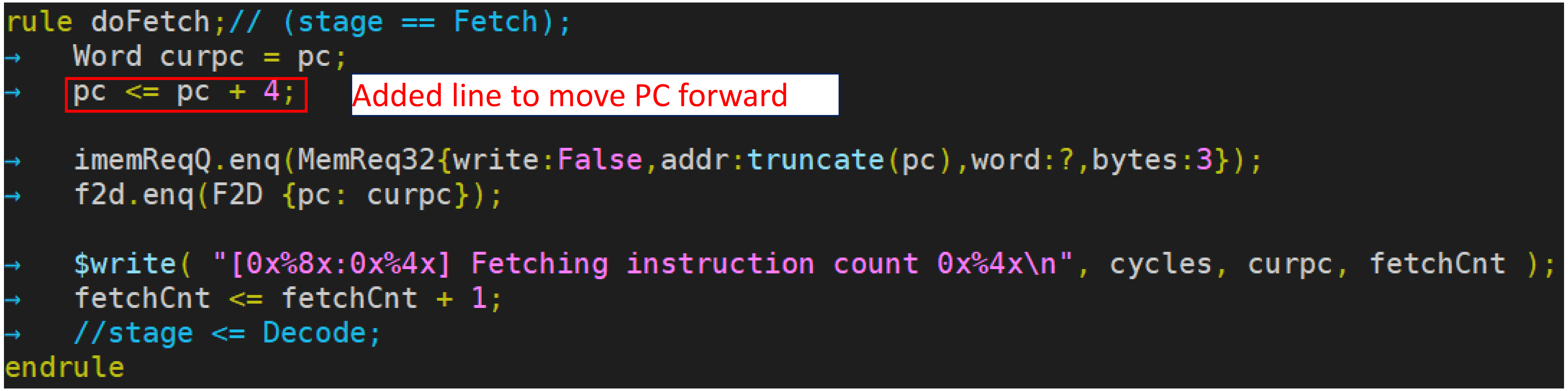
```
[0x00002134:0x0368] Fetching instruction count 0x002f
[0x00002134:0x0368] Executing
[0x00002134:0x0368]              Mem read from 0x00000ca8
[0x00002134:0x0364] Writeback writing 00001950 to 15
[0x00002137:0x0368] decoding 0x0007c703
[0x00002138:0x036c] Fetching instruction count 0x0030
[0x00002138:0x0368] decoding 0x0007c703
[0x00002138:0x0368] Executing
[0x00002138:0x0368]              Mem read from 0x00001950
[0x00002139:0x036c] Fetching instruction count 0x0031
[0x0000213c:0x036c] decoding 0x000017b7
[0x0000213c:0x0368] Writeback writing 000000aa to 14
[0x0000213d:0x036c] Fetching instruction count 0x0032
[0x0000213d:0x0368] Executing
[0x0000213d:0x0368]              Mem read from 0x00001950
[0x0000213e:0x036c] decoding 0x000017b7
[0x0000213f:0x036c] Fetching instruction count 0x0033
[0x00002141:0x0368] Writeback writing 000000aa to 14
[0x00002142:0x036c] Executing
[0x00002143:0x036c] decoding 0x000017b7
[0x00002144:0x0370] Fetching instruction count 0x0034
```

Execution hangs before reaching end!

Same instruction loaded multiple times!

Step 2: Predict PC + 4

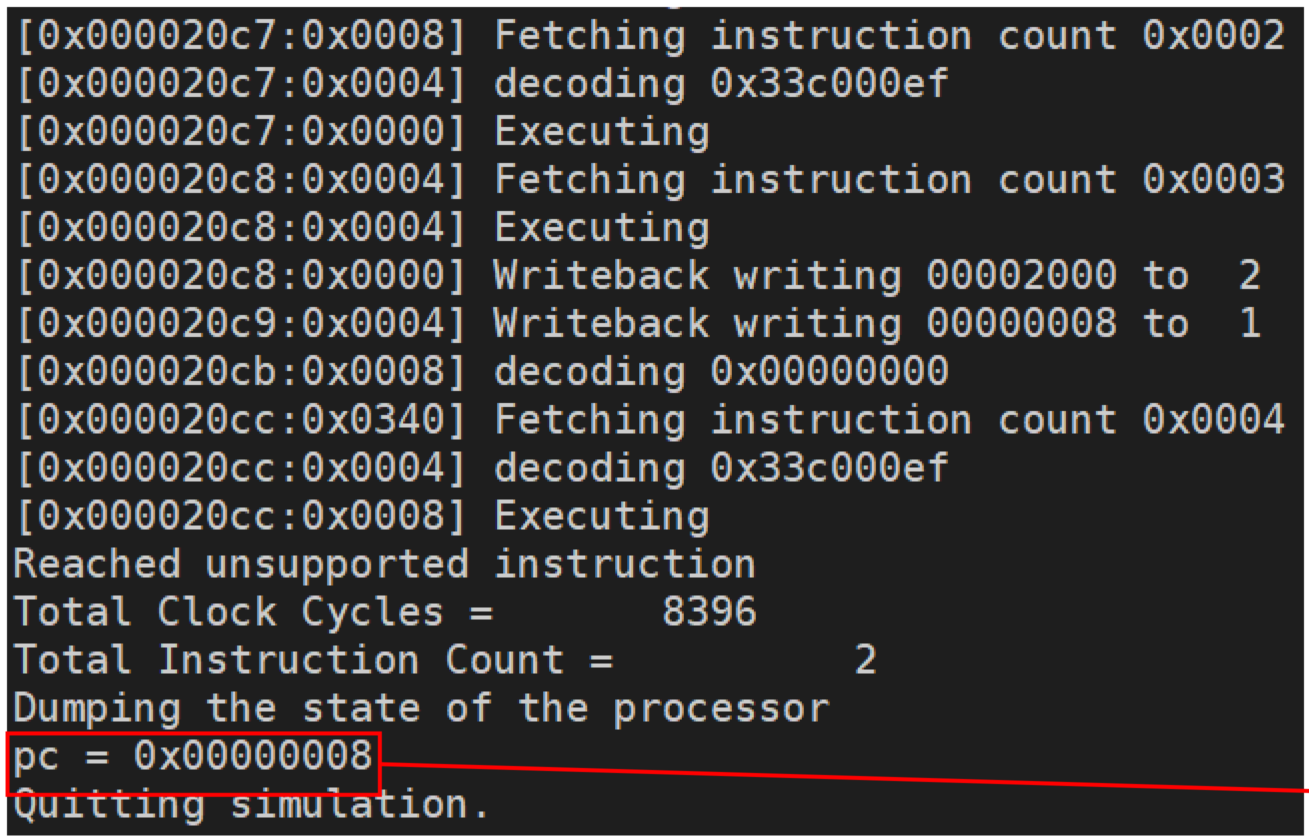
- Keep moving PC forward, predicting PC+4 every time



```
rule doFetch;// (stage == Fetch);
→   Word curpc = pc;
→   pc <= pc + 4;
→   imemReqQ.enq(MemReq32{write:False,addr:truncate(pc),word:?,bytes:3});
→   f2d.enq(F2D {pc: curpc});

→   $write( "[0x%8x:0x%4x] Fetching instruction count 0x%4x\n", cycles, curpc, fetchCnt );
→   fetchCnt <= fetchCnt + 1;
→   //stage <= Decode;
endrule
```

Did that work?

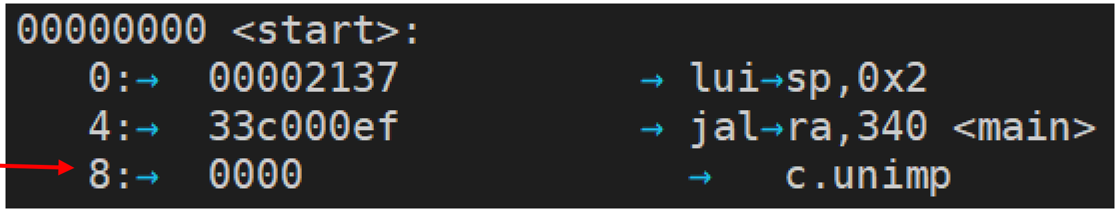
- Encounters unsupported instruction after two instructions!



```
[0x000020c7:0x0008] Fetching instruction count 0x0002
[0x000020c7:0x0004] decoding 0x33c000ef
[0x000020c7:0x0000] Executing
[0x000020c8:0x0004] Fetching instruction count 0x0003
[0x000020c8:0x0004] Executing
[0x000020c8:0x0000] Writeback writing 00002000 to  2
[0x000020c9:0x0004] Writeback writing 00000008 to  1
[0x000020cb:0x0008] decoding 0x00000000
[0x000020cc:0x0340] Fetching instruction count 0x0004
[0x000020cc:0x0004] decoding 0x33c000ef
[0x000020cc:0x0008] Executing
Reached unsupported instruction
Total Clock Cycles =       8396
Total Instruction Count =          2
Dumping the state of the processor
pc = 0x00000008
Quitting simulation.
```

Wrongly predicted jal will not branch
Should not have executed PC=8!

We need mispredict handling

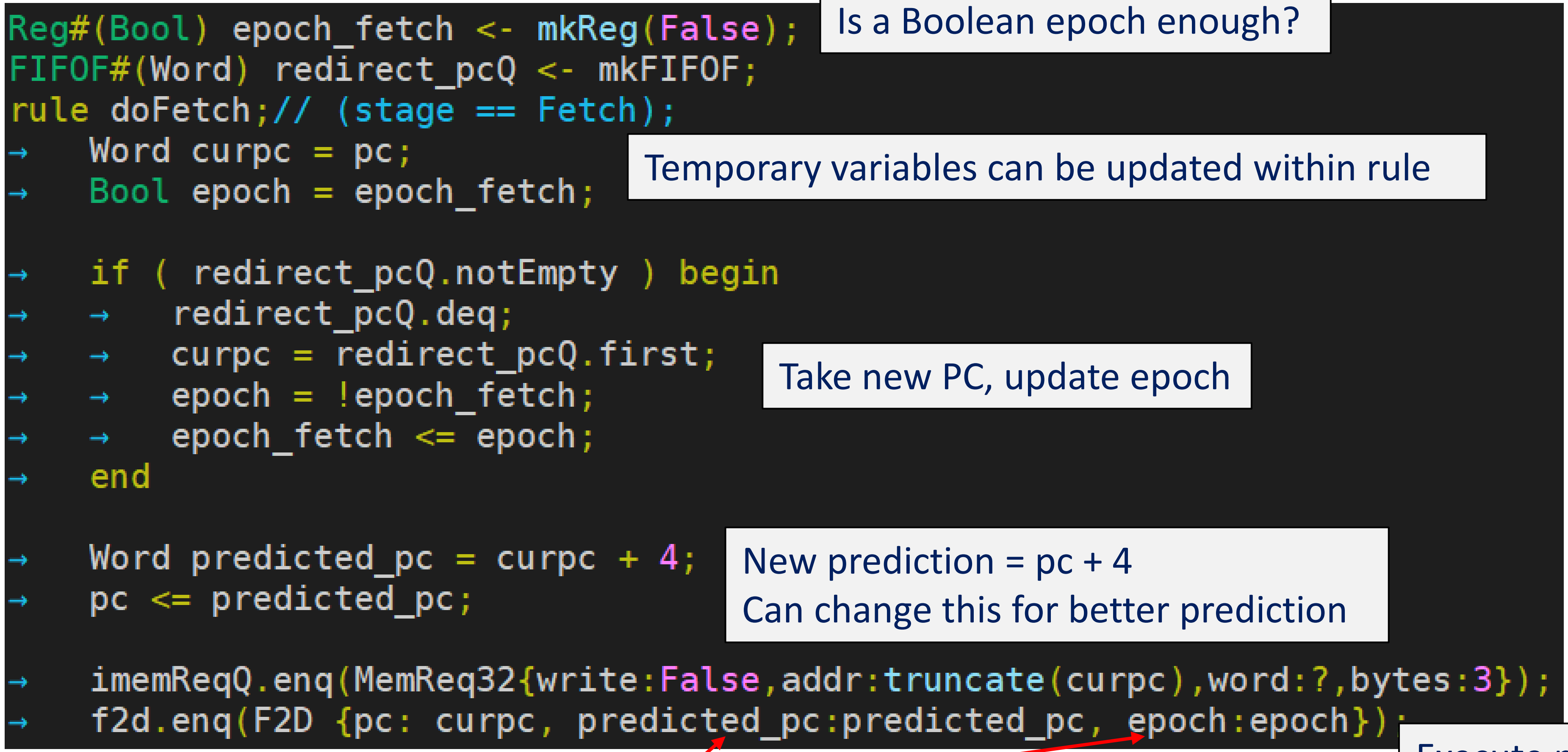


```
00000000 <start>:
   0:→   00002137          → lui→sp,0x2
   4:→   33c000ef          → jal→ra,340 <main>
   8:→   0000                →   c.unimp
```

Step 3: Solve control hazards with epochs

- Remember: Each instruction tagged with an epoch value
  - Once mispredict is detected at execute
    1. Correct PC is sent to fetch
    2. Epoch is updated
    3. Future instructions arriving at execute marked with stale epoch are ignored

Step 3: Add epochs – Fetch



```
Reg#(Bool) epoch_fetch <- mkReg(False);
FIFOF#(Word) redirect_pcQ <- mkFIFOF;
rule doFetch;// (stage == Fetch);
→   Word curpc = pc;
→   Bool epoch = epoch_fetch;

→   if ( redirect_pcQ.notEmpty ) begin
→   →   redirect_pcQ.deq;
→   →   curpc = redirect_pcQ.first;
→   →   epoch = !epoch_fetch;
→   →   epoch_fetch <= epoch;
→   end

→   Word predicted_pc = curpc + 4;
→   pc <= predicted_pc;

→   imemReqQ.enq(MemReq32{write:False,addr:truncate(curpc),word:?,bytes:3});
→   f2d.enq(F2D {pc: curpc, predicted_pc:predicted_pc, epoch:epoch});
```

Is a Boolean epoch enough?

Temporary variables can be updated within rule

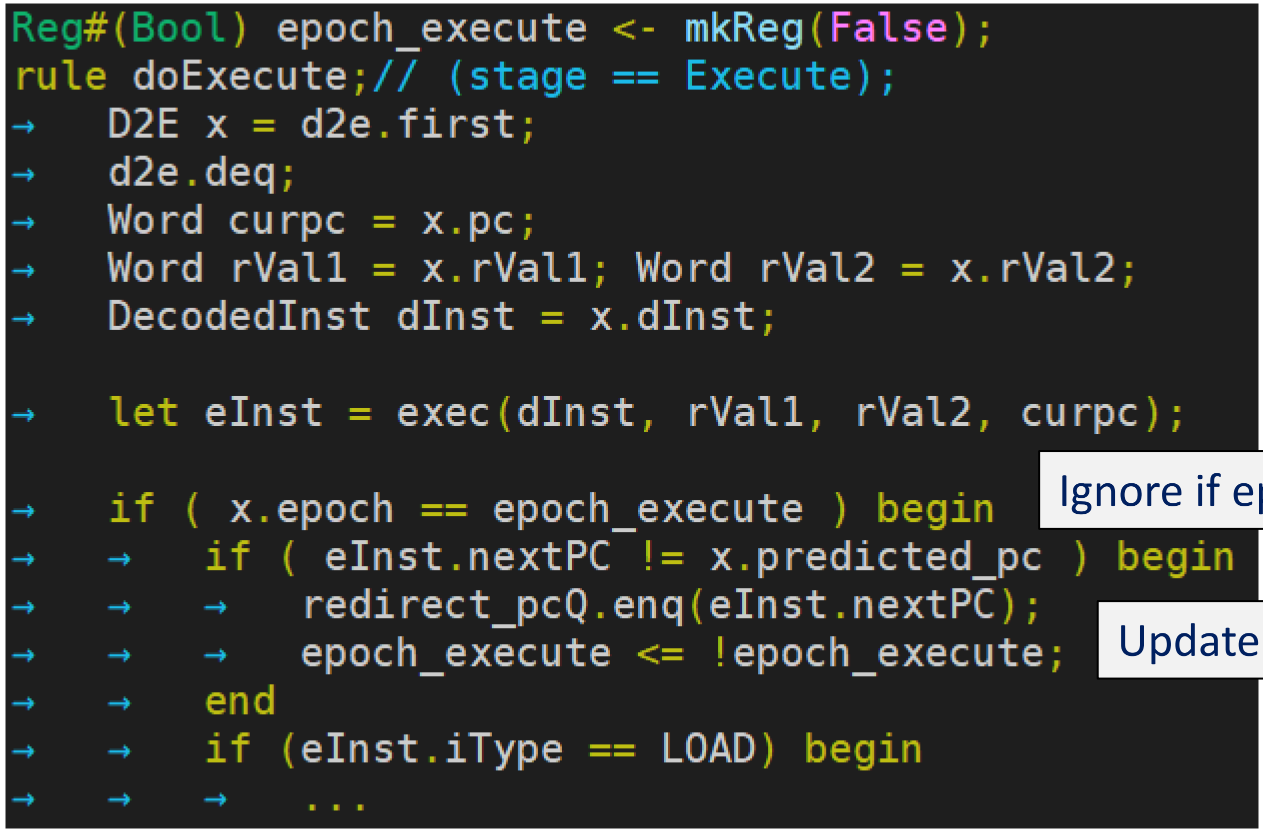
Take new PC, update epoch

New prediction = pc + 4
Can change this for better prediction

f2d needs to be augmented with predicted_pc and epoch

Execute needs to discover:
1. If prediction is correct
2. If this is from a mispredicted path

Step 3: Add epochs – Execute



```
Reg#(Bool) epoch_execute <- mkReg(False);
rule doExecute;// (stage == Execute);
→   D2E x = d2e.first;
→   d2e.deq;
→   Word curpc = x.pc;
→   Word rVal1 = x.rVal1; Word rVal2 = x.rVal2;
→   DecodedInst dInst = x.dInst;

→   let eInst = exec(dInst, rVal1, rVal2, curpc);

→   if ( x.epoch == epoch_execute ) begin
→   →   if ( eInst.nextPC != x.predicted_pc ) begin
→   →   →   redirect_pcQ.enq(eInst.nextPC);
→   →   →   epoch_execute <= !epoch_execute;
→   →   end
→   →   if (eInst.iType == LOAD) begin
→   →   →   ...
```

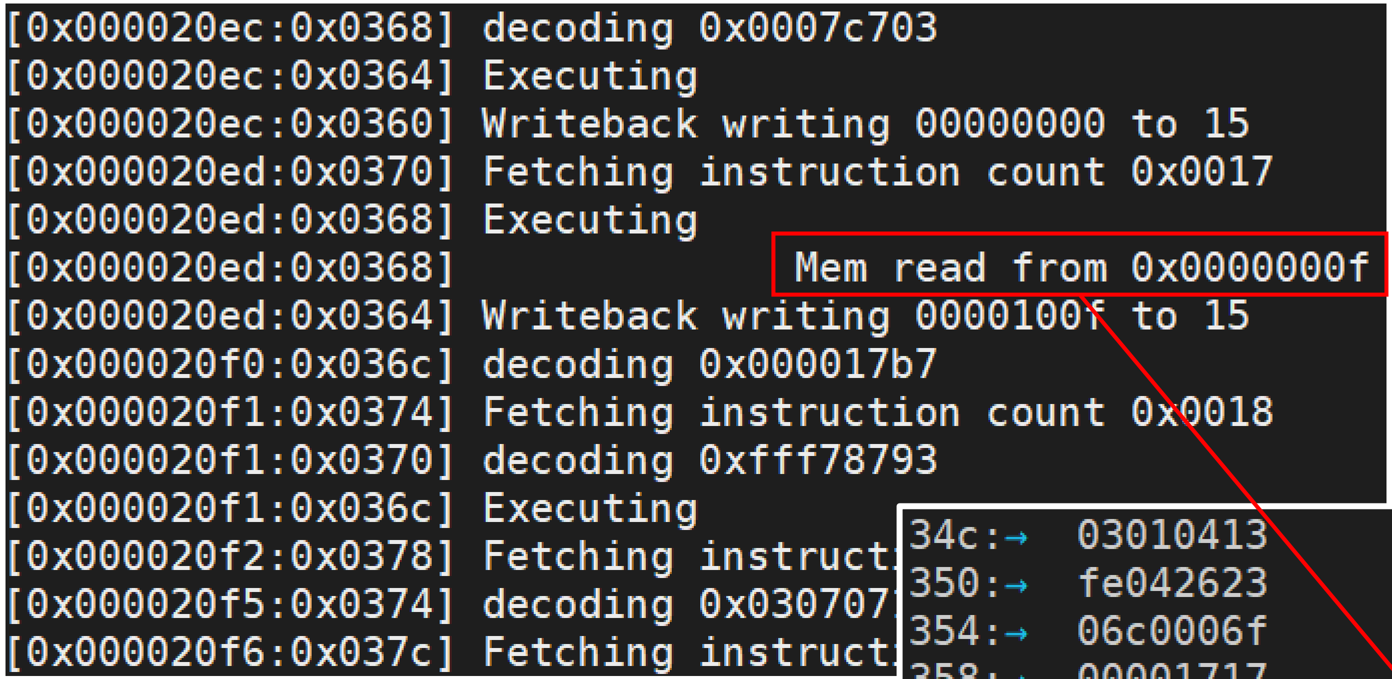
Ignore if epoch is wrong

Update epoch, send new PC if prediction is wrong

Note: d2e also must be augmented with epoch and predicted_pc

Did that work?

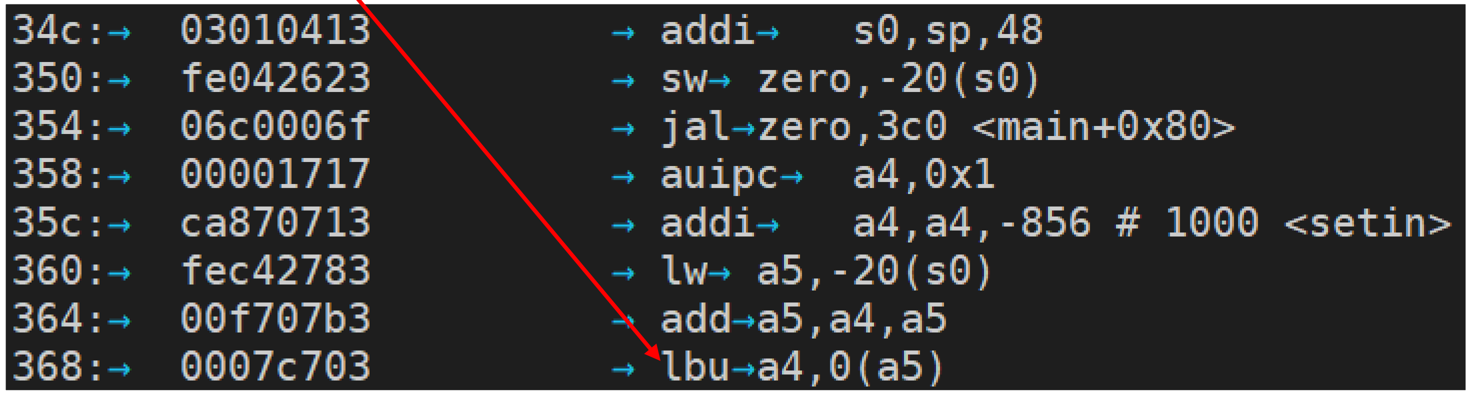
- Hangs…



```
[0x000020ec:0x0368] decoding 0x0007c703
[0x000020ec:0x0364] Executing
[0x000020ec:0x0360] Writeback writing 00000000 to 15
[0x000020ed:0x0370] Fetching instruction count 0x0017
[0x000020ed:0x0368] Executing
[0x000020ed:0x0368]              Mem read from 0x0000000f
[0x000020ed:0x0364] Writeback writing 0000100f to 15
[0x000020f0:0x036c] decoding 0x000017b7
[0x000020f1:0x0374] Fetching instruction count 0x0018
[0x000020f1:0x0370] decoding 0xfff78793
[0x000020f1:0x036c] Executing
[0x000020f2:0x0378] Fetching instruct
[0x000020f5:0x0374] decoding 0x030707
[0x000020f6:0x037c] Fetching instruct
```

Mem read from program memory!
The current system does not support dmem read from instruction memory

Data hazard!



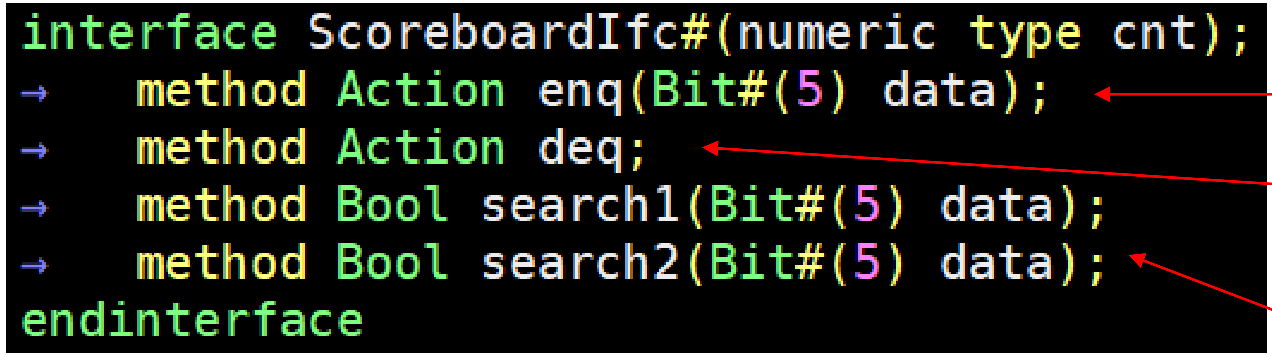
```
34c:→   03010413          → addi→   s0,sp,48
350:→   fe042623          → sw→ zero,-20(s0)
354:→   06c0006f          → jal→zero,3c0 <main+0x80>
358:→   00001717          → auipc→   a4,0x1
35c:→   ca870713          → addi→   a4,a4,-856 # 1000 <setin>
360:→   fec42783          → lw→ a5,-20(s0)
364:→   00f707b3          → add→a5,a4,a5
368:→   0007c703          → lbu→a4,0(a5)
```

Step 4: Solving data hazards

- Part 1: Stalling
  - How to detect data hazards?
  - The decode stage must know whether a previous instruction incurs data hazard
    - Previous instruction in flight will write to a register I need to read from?
  - Restriction: Detection must happen combinationally, within the decode cycle
    - Otherwise, we will slow down the pipeline
    - Or, break down decode into multiple pipeline stages
- Part2: Forwarding
  - To be continued

Detecting data hazards: Scoreboard

- Module which keeps track of destination registers
  - Decode inserts the destination register number (if any)
  - Writeback removes oldest target
  - Decode checks if any source registers exist in scoreboard, stall if so
- Interface of scoreboard:



```
interface ScoreboardIfc#(numeric type cnt);
    method Action enq(Bit#(5) data);
    method Action deq;
    method Bool search1(Bit#(5) data);
    method Bool search2(Bit#(5) data);
endinterface
```

Insert destination register number

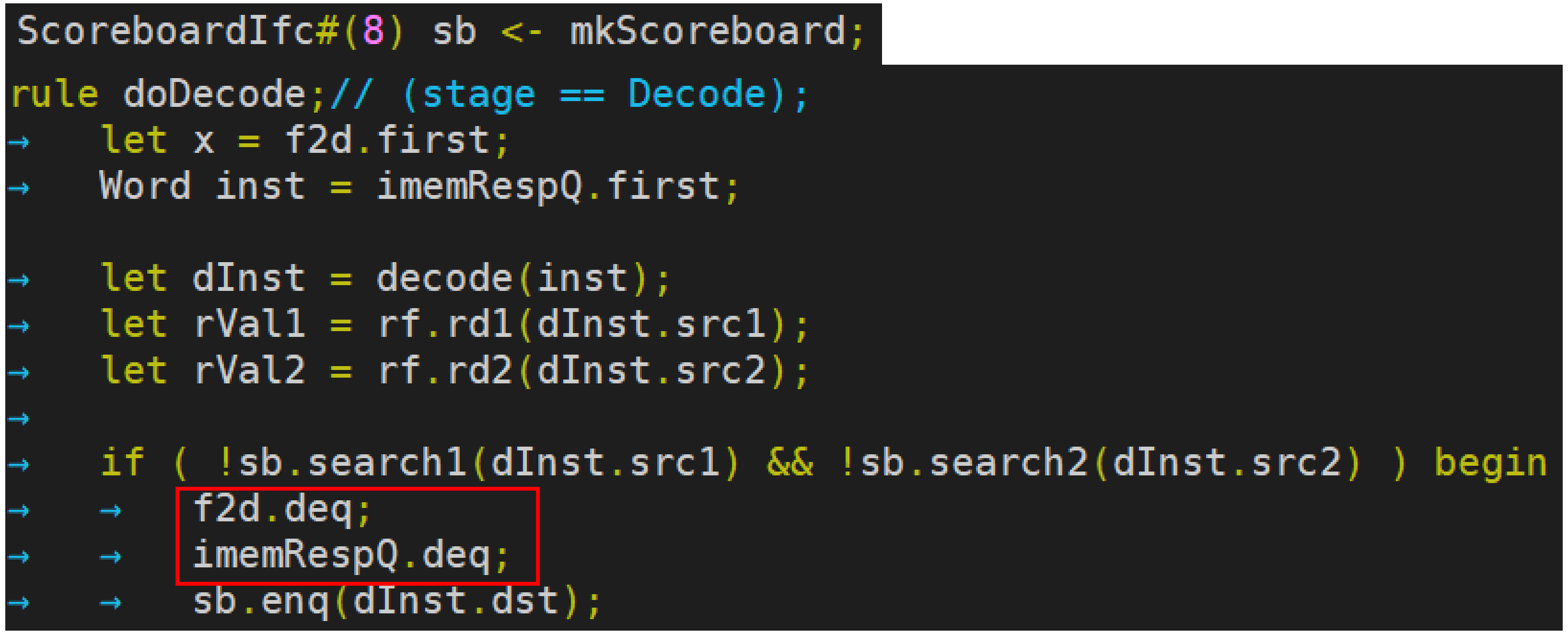
Remove oldest target

Two search methods for checking maximum of two input operands

Why do we need two separate methods?
Both searches need to happen in same cycle!

Decode stage for correct stalling

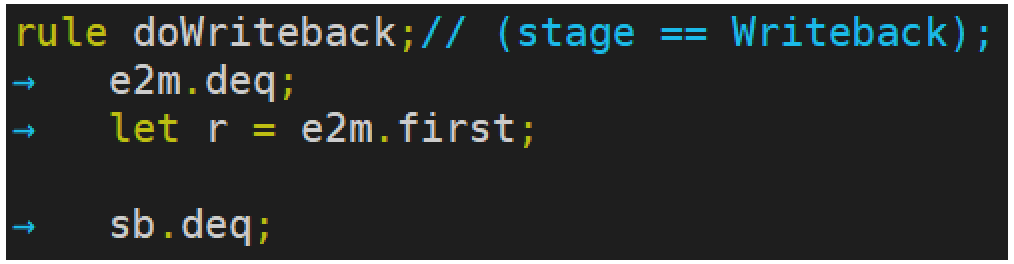
- ❑ Stall unless both input operands are not found in scoreboard
  - if (!sb.search1(dInst.src1) && !sb.search2(dInst.src2)) begin
  - f2d.deq and imemRespQ.deq should only be done when not stalling!
- ❑ When not stalling, insert destination register into scoreboard
  - sb.enq(dInst.dst)



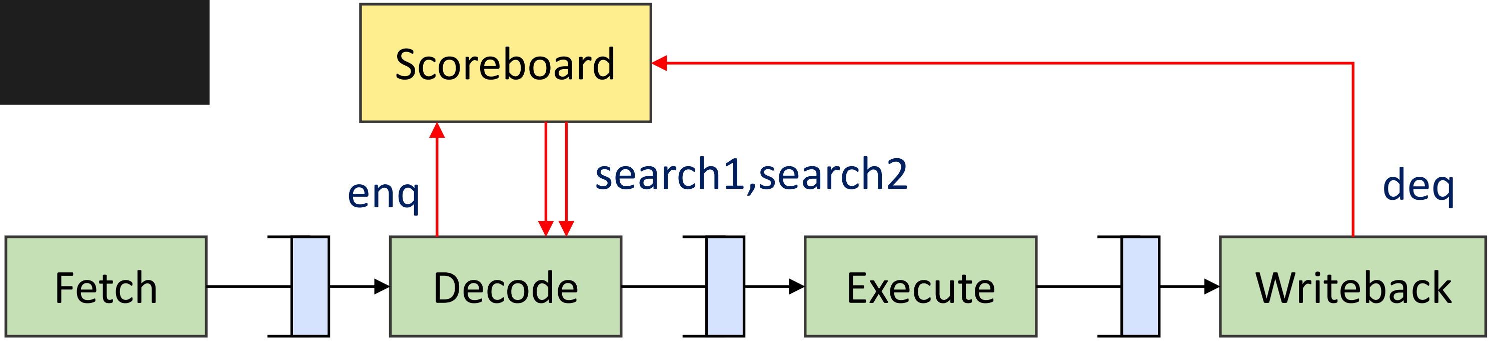
```
ScoreboardIfc#(8) sb <- mkScoreboard;
rule doDecode;// (stage == Decode);
    let x = f2d.first;
    Word inst = imemRespQ.first;

    let dInst = decode(inst);
    let rVal1 = rf.rd1(dInst.src1);
    let rVal2 = rf.rd2(dInst.src2);

    if ( !sb.search1(dInst.src1) && !sb.search2(dInst.src2) ) begin
        f2d.deq;
        imemRespQ.deq;
        sb.enq(dInst.dst);
```

Writeback stage for correct stalling

- Writeback should remove the current instruction’s dst from scoreboard
  - All instructions are in-order, so simply removing the oldest works
  - call “sb.deq”

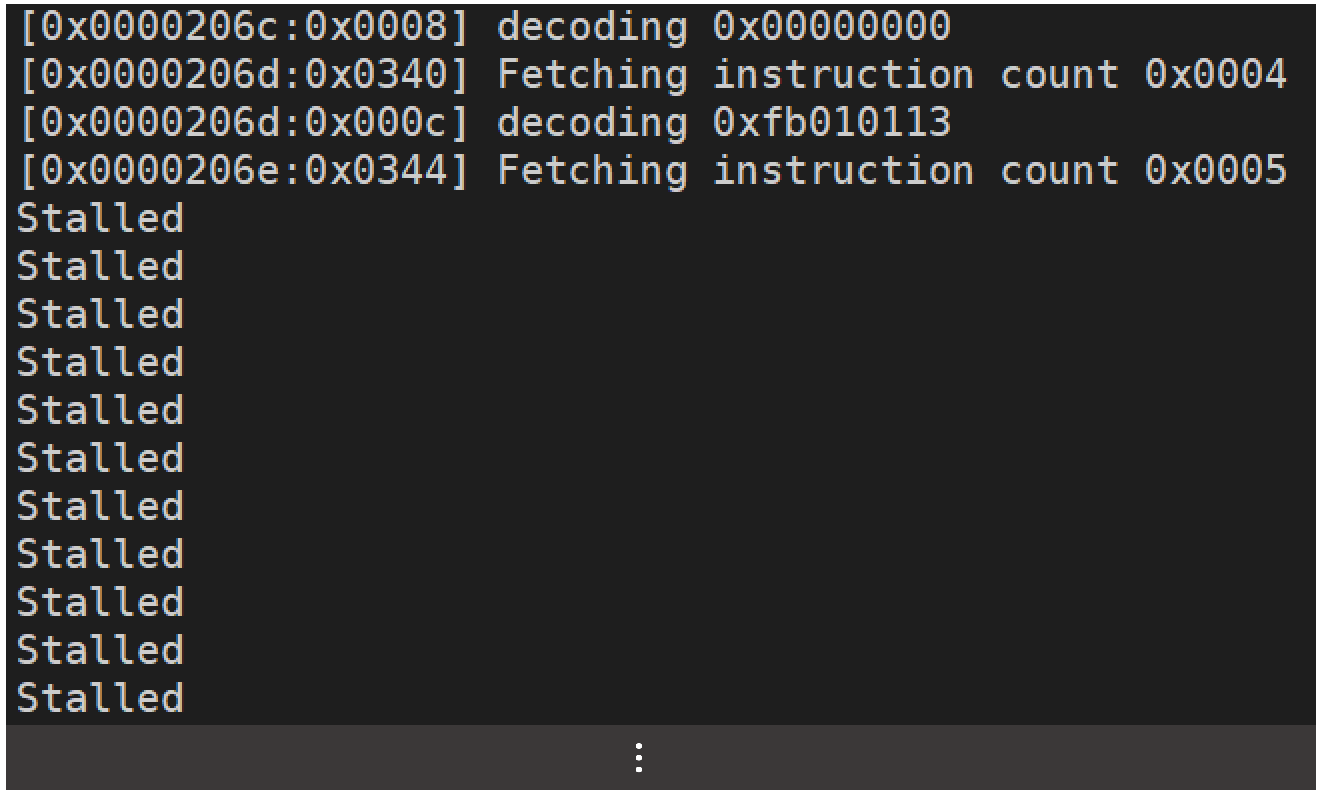


```
rule doWriteback;// (stage == Writeback);
→   e2m.deq;
→   let r = e2m.first;

→   sb.deq;
```



Does this work?

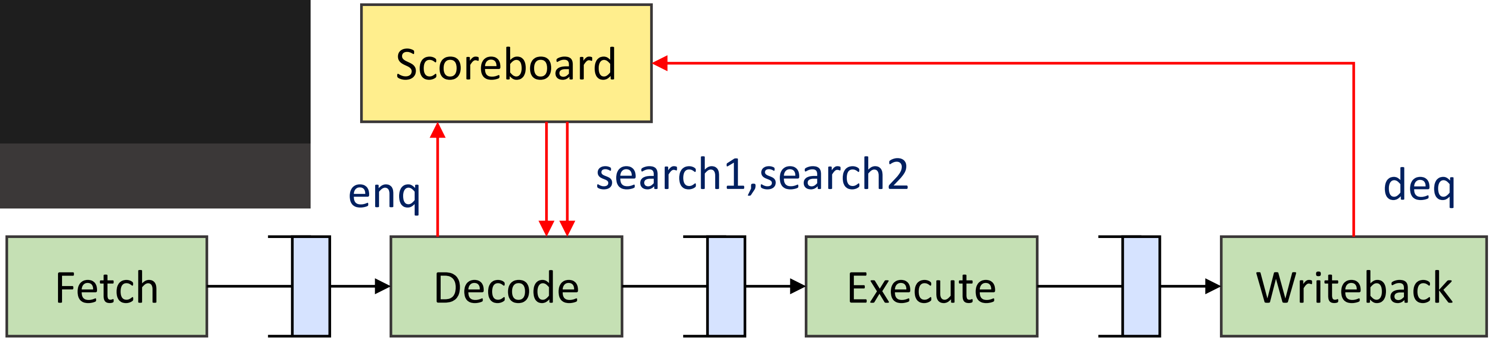
- Stalls forever… We are not deq'ing some things we enq'd!



```
[0x0000206c:0x0008] decoding 0x00000000
[0x0000206d:0x0340] Fetching instruction count 0x0004
[0x0000206d:0x000c] decoding 0xfb010113
[0x0000206e:0x0344] Fetching instruction count 0x0005
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
Stalled
⋮
```

We only deq sb in writeback!
Some instructions don't reach writeback!
(doExecute doesn't push into e2m)

- Epoch mismatch
- STORE instructions, …



Continuing Step 4: Data hazards

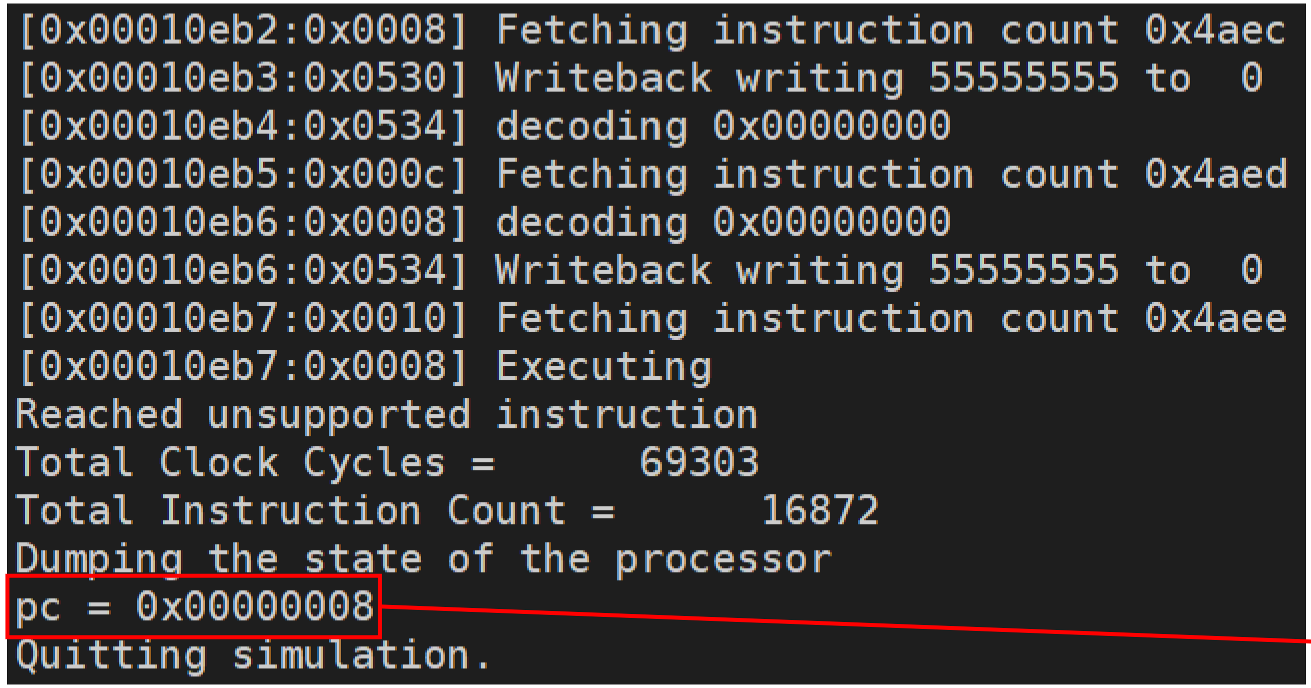
- Do we put sb.deq in execute as well?
  - No! sb has in-order semantics,
  - if execute and writeback try to deq at the same time, incorrect behavior…
- All instructions arriving at doExecute should enq ***something*** into e2m
  - Even if, say misprediction detected via epochs

  - sb.deq only in doWriteback
  - Should not wait for memory, should not write anything to rf
  - isMem = False, dst = 0

Does this work?

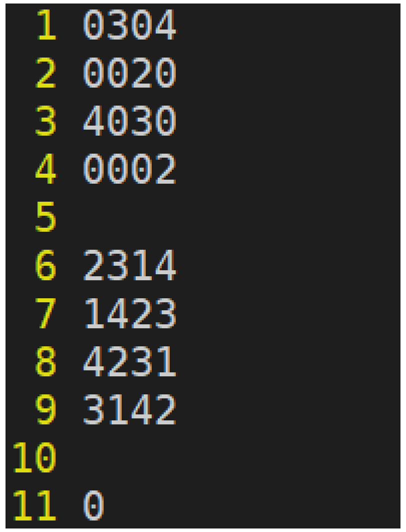
- ❑ Yes! Finally correct results!
- ❑ How is performance? Can we do better?

system.log

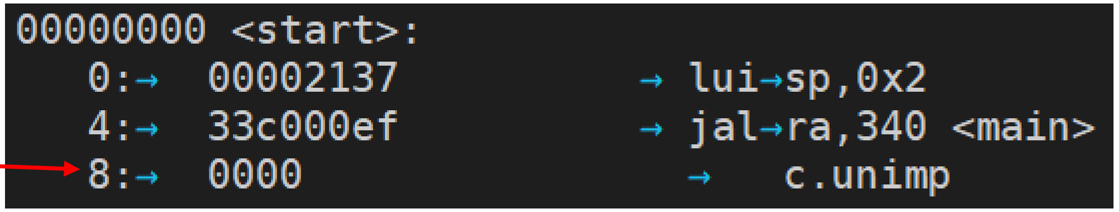


```
[0x00010eb2:0x0008] Fetching instruction count 0x4aec
[0x00010eb3:0x0530] Writeback writing 55555555 to  0
[0x00010eb4:0x0534] decoding 0x00000000
[0x00010eb5:0x000c] Fetching instruction count 0x4aed
[0x00010eb6:0x0008] decoding 0x00000000
[0x00010eb6:0x0534] Writeback writing 55555555 to  0
[0x00010eb7:0x0010] Fetching instruction count 0x4aee
[0x00010eb7:0x0008] Executing
Reached unsupported instruction
Total Clock Cycles =      69303
Total Instruction Count =      16872
Dumping the state of the processor
pc = 0x00000008
Quitting simulation.
```

output.log



```
0304
0020
4030
0002

2314
1423
4231
3142

0
```



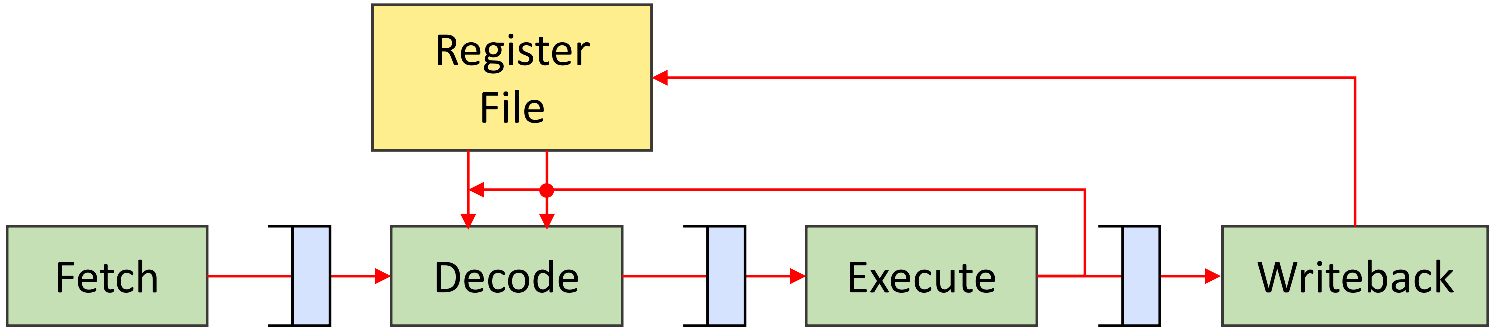
```
00000000 <start>:
   0:→   00002137          → lui→sp,0x2
   4:→   33c000ef          → jal→ra,340 <main>
   8:→   0000                →   c.unimp
```

Things to solve

1. Branch hazard – Done!
2. Load-Use hazard – Stalling
3. Read-After-Write hazard – Stalling, Forwarding
    - Pipeline is correct already, but now to improve performance!

Implementing forwarding

- Add a ***combinational*** forwarding path from execute to decode
  - If the current cycle's execute results can be used as one of inputs of decode, use that value
- Regardless of whether scoreboard.search1/2 returns true or false,
  If forward path has a source operand, we can use that value and not stall



Aside: Inter-rule combinational communication in Bluespec

- So far, communication between rules have been via state
  - Registers, FIFOs
  - State updates only become visible at the next cycle!
  - How do we make doExecute send bypass information to doDecode combinationally?
- Solution: “Wires”
  - Used just like Bluespec Registers, except data is available in the same clock cycle
  - Data is not stored across clock cycles
  - Many types, but easiest is “mkDWire”
    - Provide a “default” value, which will be read if the wire is not written to within that cycle

```
Wire#(Bit#(32)) wireA <- mkDWire(32'hffffffff);
```

32 bit wire with default value of 0xffffffff

Aside: Inter-rule combinational communication in Bluespec

- ❑ Execute stage should provide two values
  - ○ Destination register index, and its new value
  - ○ Create a wire that can combinationally send
    - Default value is for the zero register, since zero register value is always zero

```
typedef struct {
→   RIndx dst;
→   Word data;
} BypassTarget deriving(Bits,Eq);
```

```
Wire#(BypassTarget) forwardE <- mkDWire(BypassTarget{dst:0,data:0});
```

In Execute

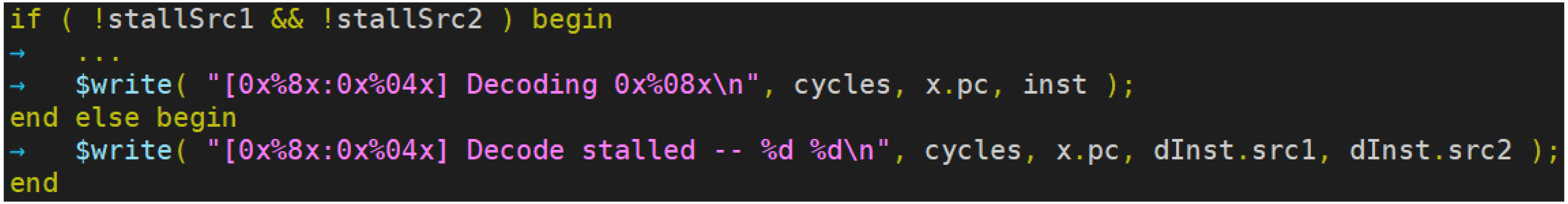
```
forwardE <= BypassTarget{dst:eInst.dst, data:eInst.data};
```

In Decode

```
Bool stallSrc1 = sb.search1(dInst.src1);
Bool stallSrc2 = sb.search2(dInst.src2);
if ( forwardE.dst > 0 ) begin
→   if ( forwardE.dst == dInst.src1 ) begin
→   →   stallSrc1 = False;
→   →   rVal1 = forwardE.data;
→   end
→   if ( forwardE.dst == dInst.src2 ) begin
```

How fast is it now?

- ❑ Add some debug output for counting stall cycles



```
if ( !stallSrc1 && !stallSrc2 ) begin
→   ...
→   $write( "[0x%8x:0x%04x] Decoding 0x%08x\n", cycles, x.pc, inst );
end else begin
→   $write( "[0x%8x:0x%04x] Decode stalled -- %d %d\n", cycles, x.pc, dInst.src1, dInst.src2 );
end
```

Count stall cycles with: `cat system.log | grep stalled | wc -l`

Question: How much faster is it now? How many milliseconds?

Some more details of current forwarding implementation

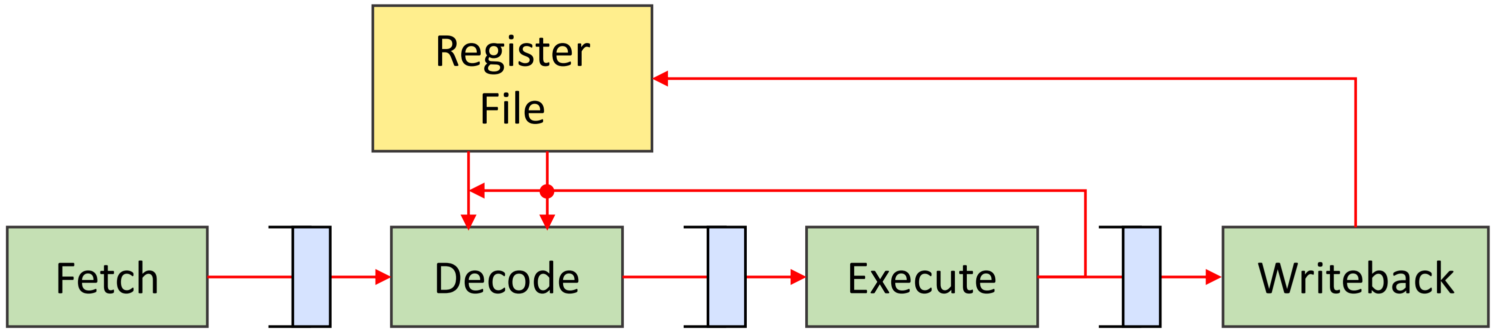
Some microbenchmark

```
 0:  40000313      addi   x6,x0,1024
 4:  00001297      auipc  x5,0x1
 8:  ffc28293      addi   x5,x5,-4
 c:  0002a483      lw     x9,0(x5)
10:  0042a903      lw     x18,4(x5)
14:  012489b3      add    x19,x9,x18
18:  01332023      sw     x19,0(x6)
1c:  c0001073      unimp
```

Why did this stall?

```
…
[0x00000005:0x0010] Decode stalled --  5  0
[0x00000005:0x0008] Writeback writing 00001000 to  5
[0x00000006:0x0010] Decoding 0x0042a903
[0x00000006:0x000c] Writeback writing 00000001 to  9
[0x00000007:0x0018] Fetching instruction count 0x0006
[0x00000007:0x0010]             Mem read from 0x00001004
[0x00000007:0x0010] Executing
[0x00000007:0x0014] Decode stalled --  9 18
[0x00000008:0x0014] Decode stalled --  9 18
…
```

Load-use hazard must stall



Why did instruction 0x10 stall?

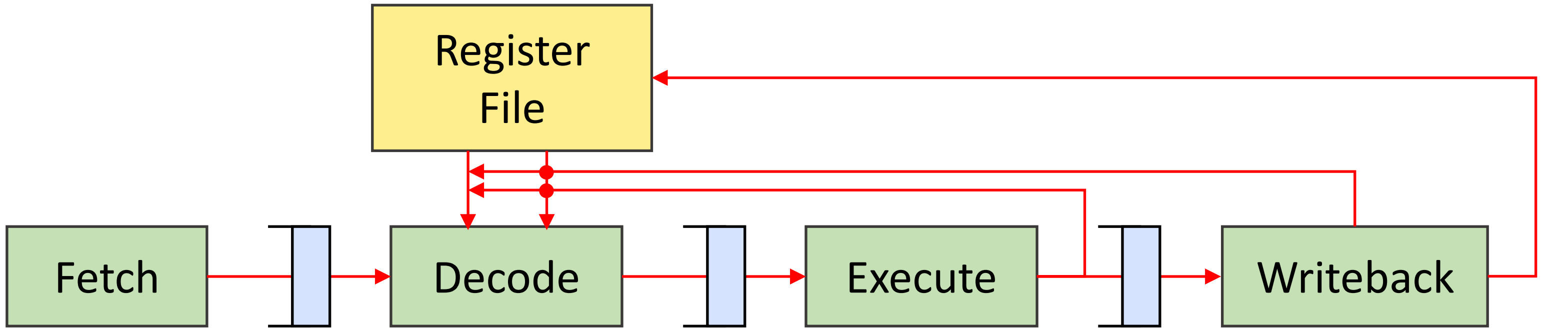
A more complete forwarding solution

- ❑ Writeback needs a forwarding path too!
- ❑ x5 is available from register file after Writeback of addi
  - ○ An instruction dependent (lw) on x5 which is in decode while addi is in Writeback must stall
- ❑ If we add a second forwarding path, we can remove a stall cycle
  - ○ Worth it? Maybe!
  - ○ Needs benchmarking!

Microbenchmark

```
 0:  40000313  addi   x6,x0,1024
 4:  00001297  auipc  x5,0x1
 8:  ffc28293  addi   x5,x5,-4
 c:  0002a483  lw     x9,0(x5)
10:  0042a903  lw     x18,4(x5)
14:  012489b3  add    x19,x9,x18
18:  01332023  sw     x19,0(x6)
1c:  c0001073  unimp
```

2-cycle gap



The overall performance at this point

- If you have followed along to this point
  - IPC ~= 0.25
  - Clock speed…? Which of our modifications had the biggest impact on clock speed?
  - Total time…?

  - Were our decisions good ones?

- IPC is still not good!
  - What is the reason? (Best guess is fine!) – Mispredicts? Data hazards?
  - Will some of our later topics address this?