CS152: Computer Systems Architecture
Hands-On Processor Development

Sang-Woo Jun
2023

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

RTL Design

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

Simulation

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

“Tapeout”

GDSII/OASIS format sent to foundry, receive first spin chip in a few months
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!
Toolchains for FPGA development

- Typically vendor-specific
  - Xilinx: Vivado, Vitis
  - Intel/Altera: Quartus
  - Lattice: Diamond

- Robust open-source projects
  - Yosys, nextpnr, arachne-pnr, 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, ...
    - Registers (state), and combinational logic
    - Efficient, difficult to program
  - “High-Level Synthesis”: Uses familiar software programming languages
    - C-to-gates, OpenCL, ...
    - Typically compiles to Verilog/VHDL
    - Easy to program, inefficient
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?”

CPU Time = Instruction Count × CPI × 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

If core count/memory allowance needs changing...
Getting started

Change core/memory assignment if necessary
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!
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
Example simulation execution

From the simulation, we can measure the cycle count...

Performance numbers!

IPC = 16,596 / 135,944 ~= 0.122
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
- \((\text{cycle count})/\text{(clock frequency)} = \text{time to completion}!\)

- In our previous example, 135,944 cycles / 69.80 MHz = 0.0019s
  - Is this good?
  - We can do MUCH better!
CS152: Computer Systems Architecture
A Very Short Introduction to Bluespec

Sang-Woo Jun
2023

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
Bluespec System Verilog (BSV) High-Level

- Everything organized into “Modules”
  - Modules have an “interface” which other modules use to access state
  - A Bluespec model is a single top-level module consisting of other modules, etc

- Modules consist of state (other modules) and behavior
  - State: Registers, FIFOs, RAM, ...
  - Behavior: Rules, Interface
Peek into a RISC-V processor in Bluespec

Processor.bsv

interface ProcessorIfc;
> method ActionValue#(MemReq32) memReq;
> method Action memResp(Word data);
endinterface

module mkProcessor(ProcessorIfc);
> Reg#(Word) pc <- mkReg(0);
> RFile2R1W  rf <- mkRFile2R1W;
> MemorySystemIfc mem <- mkMemorySystem;
> rule doFetch (stage == Fetch);
>  > let next_pc = pc + 4;

Top.bsv

module mkTop(Empty);
> ProcessorIfc proc <- mkProcessor;

::
Greatest Common Divisor Example

- Euclid’s algorithm for computing the greatest common divisor (GCD)

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>3</td>
<td>6</td>
<td>subtract</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>swap</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>subtract</td>
</tr>
<tr>
<td>0</td>
<td>3</td>
<td>subtract</td>
</tr>
</tbody>
</table>

Answer
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);

rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
endrule

rule step2 ((x <= y) && (y != 0));
  y <= y-x;
  if (y-x == 0) begin
    outQ.enq(x);
  end
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
endmethod

method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
endmodule
module mkGCD (GDCIfc);
  Reg#(Bit#(32)) x <- mkReg(0);
  Reg#(Bit#(32)) y <- mkReg(0);
  FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);
endmodule

rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
endrule
rule step2 ((x <= y) && (y != 0));
  y <= y - x;
  if (y - x == 0) begin
    outQ.enq(x);
  end
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
endmodule
module mkGCD (GDCIfc);

Reg#(Bit#(32)) x <- mkReg(0);
Reg#(Bit#(32)) y <- mkReg(0);
FIFOF#(Bit#(32)) outQ <- mkSizedFIFOF(2);

rule step1 ((x > y) && (y != 0));
  x <= y; y <= x;
endrule

rule step2 ((x <= y) && (y != 0));
  y <= y - x;
  if (y - x == 0) begin
    outQ.enq(x);
  end
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (y == 0);
  x <= a; y <= b;
endmethod

method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
endmodule

State

Rules (Behavior)

Interface (Behavior)

Interface methods are also atomic transactions
Can be called only when guard is satisfied
When guard is not satisfied, rules that call it cannot fire
Bluespec Modules – Interface

- Modules encapsulates state and behavior (think C++/Java classes)
- Can be interacted from the outside using its “interface”
  - Interface definition is separate from module definition
  - Many module definitions can share the same interface: Interchangeable implementations
- Interfaces can be parameterized
  - Like C++ templates “FIFO#(Bit#(32))”
  - Not important right now

```verbatim
interface GDCIfc;
  method Action start(Bit#(32) a, Bit#(32) b);
  method ActionValue#(Bit#(32)) result();
endinterface

module mkGCD (GDCIfc);
  ...
  method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
    x <= a; y <= b;
  endmethod
  method ActionValue#(Bit#(32)) result();
    outQ.deq;
    return outQ.first;
  endmethod
endmodule
```
Bluespec Module – Interface Methods

❑ Three types of methods
  o Action : Takes input, modifies state
  o Value : Returns value, does not modify state
  o ActionValue : Returns value, modifies state

❑ Methods can have “guards”
  o Does not allow execution unless guard is True

```
rule ruleA;
  moduleA.actionMethod(a,b);
  Int#(32) ret = moduleA.valueMethod(c,d,e);
  Int#(32) ret2 <= moduleB.actionValueMethod(f,g);
endrule

method Action start(Bit#(32) a, Bit#(32) b) if (y==0);
  x <= a; y <= b;
endmethod
method ActionValue#(Bit#(32)) result();
  outQ.deq;
  return outQ.first;
endmethod
```

Note the “<-” notation

Guard

Automatically introduces “implicit guard” if outQ is empty
Combinational circuits in Bluespec: Rules

- A Bluespec rule represents a state transfer via combinational circuits
  - Much like Verilog “always” and VHDL “process”
  - Can call methods of other modules
    - e.g., outQ.enq – Introduces implicit guard if outQ is full

```plaintext
rule step2 ((x <= y) && (y != 0));

    y <= y-x;
    if ( y-x == 0 ) begin
        outQ.enq(x);
    end
endrule
```

“enq” encapsulates more combinational logic
Combinational circuits in Bluespec: Functions

- Functions are combinational – do not allow state changes
  - Can be defined within or outside module scope
  - No state change allowed, only performs computation and returns value

```plaintext
// Function example
function Int#(32) square(Int#(32) val);
  return val * val;
endfunction

rule rule1;
  x <= square(12);
endrule
```

Combinational ALU implemented using a function
Bluespec Rules Are Atomic Transactions

- Only has access to state values from before rule began firing
- State update happens once as the result of rule firing
  - e.g.,
    ```
    // x == 0, y == 1
    x <= y; y <= x; // x == 1, y == 0
    ```
  - e.g.,
    ```
    // x == 0, y == 1
    x <= 1; x <= y; // write conflict error!
    ```

Intuition: All statements in rule execute in parallel
Bluespec State – FIFO

- Fixed size queue
- Parameterized interface with guarded methods
  - e.g., testQ.enq(data); // Action method. Blocks when full
  - testQ.deq; // Action method. Blocks when empty
  - dataType d = testQ.first; // Value method. Blocks when empty
- FIFOF adds two more methods
  - testQ.notEmpty returns bool
  - testQ.notFull returns bool
- Provided as library
  - Needs “import FIFO::*;” at top

```verilog
FIFO#(Bit#(32)) testQ <- mkSizedFIFO(2);
rule enqdata; // whole rule does not fire if testQ is full
  if ( x ) y <= z;
  testQ.enq(32'h0);
endrule
```
Bluespec rules:
State and temporary variables

- **State**: Defined outside rules, data stored across clock cycles
  - All state updates happen atomically
  - Reg#(...), FIFO#(...)  
  - **Register state assignment uses** "$\leftarrow\$"

- **Temporary variables**: Defined within rules, data local to a rule execution
  - Intuition: Rule-local variables
  - Follows sequential semantics similar to software languages
  - **Temporary variable value assignment uses** "$\leftarrow\$"

- **Same syntax as Verilog/VHDL**
Bluespec rules:
State and temporary variables

- Temporary variables behave as you would expect

```plaintext
Reg#(Bit#(32)) a <- mkReg(1);  // State
Reg#(Bit#(32)) b <- mkReg(4);  // State
rule rule_a;
    Bit#(32) c = a+1;  // Temporary variable c == 2
    Bit#(32) d = (c + b)/2;  // Temporary variable d == 3
    a <= d;  // State a == 3 after this cycle
    b <= a+d;  // State b == 4 after this cycle
endrule
```
Behavior of Bluespec Rules

- At every cycle, all rules that can fire, will fire
  - All guards are satisfied
  - No conflicts between rules

- Conflict between rules?
  - Two rules updating same state (writing to same register, enq’ing to same FIFO)
    - One rule enq’ing, one rule deq’ing is OK!
  - When conflict, only one rule fires
    - Typically the first one in the source file
CS152: Computer Systems Architecture
Dive Into The Example Processor

Sang-Woo Jun
2023

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!

![Hardware Platform Diagram]
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);
→ RF file2R1W rf <- mkRF file2R1W;

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

Processor

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

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
```

Only one rule can fire at a time
The fetch stage

- 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%x:0x%x4x] Fetching instruction count 0x%x4
", cycles, curpc, instCnt );
  stage <= Decode;
endrule
```

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

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

```verilog
rule doDecode (stage == Decode);
  let x = f2d.first;
  f2d.deq;
  Word inst = imemRespQ.first;
  imemRespQ.deq;
  let dInst = decode(inst);
  let rVal1 = rf.rdi(dInst.src1);
  let rVal2 = rf.rd2(dInst.src2);
  d2e.enq(D2E {pc: x.pc, dInst: dInst, rVal1: rVal1, rVal2: rVal2});
  $write("[0x%x:0x%x] 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

```haskell
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);

typedef enum {Add, Sub, And, Or, Xor, Slt, Sltu, Sll, Srl, Sra, Mul} AluFunc 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]);
    ...
}
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;
case(opcode)
  -> op0: 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

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>

E.g., add x9, x20, x21
The execute stage

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

```plaintext
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
      ...
  end
```

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

```verilog
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?
Looking at sample execution

- Try running “make runsim”
- “Mul” not part of rv32i!

```
[0x000212ee:0x0049c] Fetching instruction count 0x40db
[0x000212f2:0x0049c] decoding 0xfd42703
[0x000212f3:0x0049c] Executing
[0x000212f3:0x0049c] Mem read from 0x00001fd0
[0x000212f7:0x0049c] Writeback writing 0x00000002 to 14
[0x000212f8:0x004a0] Fetching instruction count 0x40de
[0x000212fc:0x004a0] decoding 0x02e787b3
[0x000212fd:0x004a0] Executing
Reached unsupported instruction
Total Clock Cycles = 135933
Total Instruction Count = 16604
Dumping the state of the processor
pc = 0x0000004a0
Quitting simulation.
Segmentation fault (core dumped)
```

```
498: fe442783 → lw a5,-28(s0)
49c: fdc42703 → lw a4,-28(s0)
4a0: 02e787b3 → sw a5,-28(s0)
4a4: fef42223 → mul a5,a5,a4
4a8: fe42783 → lw a5,-28(s0)
```

Don’t mind this for now

```
sw/minisudoku.dump
```

```
output.log
```

```
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

<table>
<thead>
<tr>
<th>funct7</th>
<th>rs2</th>
<th>rs1</th>
<th>funct3</th>
<th>rd</th>
<th>opcode</th>
</tr>
</thead>
<tbody>
<tr>
<td>7 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>3 bits</td>
<td>5 bits</td>
<td>7 bits</td>
</tr>
</tbody>
</table>
CS152: Computer Systems Architecture
Pipelining The Processor

Sang-Woo Jun
2023

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
    - e.g., 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 (in all rules)

```vhdl
//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

Execution hangs before reaching end!

Same instruction loaded multiple times!

Why this particular behavior?

Hint: PC update currently done in execute
Step 2: Predict PC + 4

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

```java
rule doFetch; // (stage == Fetch);
   Word curpc = pc;
   pc <= pc + 4;  // Added line to move PC forward
   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!

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

We need mispredict handling.

Dumping the state of the processor

\[ \text{pc} = 0x00000008 \]

Quitting simulation.
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

Q: Is a Boolean epoch enough?

Temporary variables can be updated within rule

Why ‘epoch’ as temporary variable?

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

```haskell
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
      ...
```
Did that work?

❑ Hangs...

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

Data hazard!
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

- Part 2: Forwarding
  - To be continued
Detecting data hazards: Scoreboard

- Module which keeps track of destination registers
  - Decode records the destination register index (if any)
  - Writeback removes oldest destination
  - Decode checks if any source registers exist in scoreboard, stall if so

- Interface of scoreboard:

```java
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)

```haskell
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);
  end
```
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”

```plaintext
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!

We only deq sb in writeback!
Some instructions don’t reach writeback!
(\text{doExecute} doesn’t push into e2m)
- Epoch mismatch
- STORE instructions, ...

```
[0x0000206c:0x0008] decoding 0x00000000
[0x0000206:d:0x0340] Fetching instruction count 0x0004
[0x0000206:d:0x000c] decoding 0xfb010113
[0x0000206e:0x0344] Fetching instruction count 0x0005
```
Continuing Step 4: Data hazards

Q: Do we put sb.deq in execute as well?
   o No! sb has in-order semantics,
   o if execute and writeback try to deq at the same time, incorrect behavior...

All instructions arriving at doExecute should enq *something* into e2m
   o Even if, say misprediction detected via epochs

   o sb.deq only in doWriteback
   o Should not wait for memory, should not write anything to rf
   o isMem = False, dst = 0
Does this work?

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

```
[0xc0001e2:0x0008] Fetching instruction count 0x4a4c
[0xc0001e3:0x00f3] Writeback writing 55555555 to 0
[0xc0001e4:0x00f3] decoding 0x00000000
[0xc0001e5:0x00c8] Fetching instruction count 0x4a4d
[0xc0001e6:0x0008] decoding 0x00000000
[0xc0001e6:0x00d] Writeback writing 55555555 to 0
[0xc0001e7:0x00a0] Fetching instruction count 0x4a4e
[0xc0001e7:0x0008] Executing
Reached unsupported instruction
Total Clock Cycles = 69303
Total Instruction Count = 16872
Dumping the state of the processor
pc = 0xc00000008
Quitting simulation.
```
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

```plaintext
Wire#(Bit#(32)) wireA <- mkDWire(32'hfffffffff);  // 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

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

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

**In Execute**

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

**In Decode**

```plaintext
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;
  → end
  → if (forwardE.dst == dInst.src2 ) begin
```

How fast is it now?

- Add some debug output for counting stall cycles

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

...[0x00000005:0x0010] Decode stalled -- 5 0
[0x00000005:0x0008] Writeback writing 00010000 to 5
[0x00000006:0x0010] Decoding 0x0042a903
[0x00000006:0x000c] Writeback writing 00000011 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

...Why did this stall?

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

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Instruction</th>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>addi x6, x0, 1024</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>auipc x5, 0x1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>addi x5, x5, -4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>c</td>
<td>lw x9, 0(x5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>lw x18, 4(x5)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>add x19, x9, x18</td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>sw x19, 0(x6)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1c</td>
<td>unimp</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The overall performance at this point

- If you have followed along to this point
  - IPC \( \approx 0.25 \)
  - 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?