Design Document Group 54

Members: William Kerr, Nathan Lakritz, Ali Hooman

Goals:

Write a RISC-V processor simulator that executes in a 5-stage pipeline with branch prediction. Support the subset of the RISC-V ISA:

- RV32I Base Instruction Set
- RV64I Base Instruction Set
- RV32M Standard Extension
- RV64M Standard Extension

We will be reusing our code from assignment 1. Must be capable of executing simple assembly language instructions.

Questions we must answer before writing code:

1. How will we implement branch prediction?
2. How will we implement forwarding?
3. How will we implement stalling for every stage?
4. How will instructions be flushed?
5. What information needs to be passed between stages?
6. Once we decode the instruction, how will we execute the instruction in another cycle?
7. Once we execute the instruction, how will we write to memory in another cycle?
8. Once we write to memory, what are we writing back, and how will we implement the writeback stage?
9. How can we accomplish reliable unit testing?

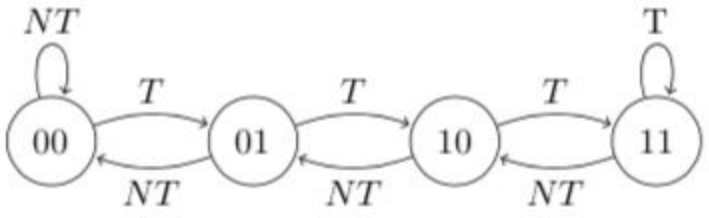
Changes from Assignment 1:

- Implement stalling & data forwarding to handle control and data hazards.
- Avoid unnecessary ALU operations on the values: Shifts and masks are OK, adds and subtracts are not.
- Implement branch prediction using a *Backwards Taken Forwards Not Taken* (BTFNT) implementation
    - Use a 32-entry *Branch Target Buffer* (BTB)
    - Implement it during the Fetch stage
- JAL instruction is now a U-type instruction instead of a UJ-type instruction.
- Unit tests are now required to make the file. They are required to be named ‘void unit_tests()’. To run them, run sim -u.

Extra Credit:

Implement your branch predictor using a 2-bit dynamic branch prediction scheme instead of the BTFNT scheme. Requires creation of a Branch History Table (BHT) in addition to the BTB.

- Branch Target Buffer (BTB)
    - Determines if instruction at PC is a branch or not
    - Determines Target of branch once Branch History Table says to take the branch
    - 32 index buffer, 2^5 unique addresses obtained from 5 lowest order bits
- Branch History Table (BHT)
    - Also referred to as a Branch Prediction Buffer
    - Maintains a history of taken and not taken branches
    - 00, 10, 01, 11 bits to track taken-ness of branch.



00 = Predict not taken, not taken at least once
10 = Predict not taken
01 = Predict taken
11 = Predict taken, taken at least once

Functions to Implement:

```
void stage_fetch(struct stage_reg_d *new_d_reg);
void stage_decode(struct stage_reg_x *new_x_reg);
void stage_execute(struct stage_reg_m *new_m_reg);
void stage_memory(struct stage_reg_w *new_w_reg);
void stage_writeback(void);
```

These pipeline functions should be considered as running simultaneously during each run of the simulator. Each function can communicate with the next only using the provided decode, execute, memory, and write stage (pipeline) registers.

Each stage register in our code represents the specific register that separates each stage in the pipeline.

- **Stage register d:** IF/ID, passes information from fetch to decode stage.
- **Stage register x:** ID/EX, passes information from decode to execute stage.
- **Stage register m:** EX/MEM, passes information from execute to memory stage.
- **Stage register w:** MEM/WB, passes information from memory to writeback stage.

The C structures for each stage register are outlined below for reference. They are stored in the riscv_sim_pipeline_framework.h header file. We can edit the fields of these structures as needed to pass sufficient values between each pipeline stage.

Provided structures:

```
struct stage_reg_d {
    uint64_t    new_pc;
    uint32_t    instruction;
};

struct stage_reg_x {
    uint64_t    new_pc;
    uint32_t    instruction;
};

struct stage_reg_m {
    uint32_t    instruction;
};

struct stage_reg_w {
    uint32_t    instruction;
};
```

We are allowed to add fields to each of these structs. We need to be careful about not adding unnecessary or redundant fields, but also take advantage of the flexibility we have here. Various flags will be extremely helpful for stalling, forwarding, and flushing. Ex. a NOP flag is a quick way to trigger a flush.

From riscv_pipeline_registers.h:

```
extern const struct stage_reg_d *   current_stage_d_register;
extern const struct stage_reg_x *   current_stage_x_register;
extern const struct stage_reg_m *   current_stage_m_register;
extern const struct stage_reg_w *   current_stage_w_register;
```

We can read values passed from other functions in previous cycles to the appropriate stage using the current_stage_?_register structs. However, we cannot write directly to the “current” register, which means all values needed for a given cycle must be transferred **before** that cycle occurs. The values may only be transferred from each stage to the following pipeline/stage register.

Example:
`stage_fetch(new_d_register)`
writes to
`current_stage_d_register.`

During the next cycle,
`stage_decode(new_x_register)`
can fetch data from
`current_stage_d_register.`

The data we’re passing between stages:

Fetch -> Decode

- The instruction and its address

Decode -> Execute

- The instruction and its address
- Type of operation
- The operands
- Where values need to be stored

Execute -> Memory

- The instruction
- Values that need to be stored or loaded
- Location in memory
- Destination register

Memory -> Writeback

- The instruction
- Values that need to be written
- Destination register

Provided functions:

The below functions are provided for manipulating the simulator within our code. These functions are all included in riscv_sim_pipeline_framework.c, do not edit this file.

- **memory_read()** - Used to pull values from program memory**.**
- **memory_write()** - Used to write values to program memory.
- **memory_initialize()** - Used in our unit test file to create simulator memory.
- **memory_status()** - Used for memory caching (pa3)
- **register_read()** - Used to read the values of registers and set them as operands either during ALU execution or memory storing
- **register_write()** - Used to write values to the destination register exclusively in the writeback stage
- **set_pc()** - Set new PC for branches and jumps.
- **get_pc()** - Used to check PC location and determine if branch is forward or backward.

```
extern bool memory_read (uint64_t address, void * value, uint64_t
size_in_bytes);
extern bool memory_write (uint64_t address, uint64_t value, uint64_t
size_in_bytes);
extern bool memory_status (uint64_t address, uint64_t *value);

extern void register_read (uint64_t register_a, uint64_t register_b,
uint64_t * value_a, uint64_t * value_b);
extern void register_write (uint64_t register_d, uint64_t value_d);

extern void     set_pc (uint64_t pc);
extern uint64_t get_pc (void);
```

Logistics:

We're making a new folder in our git repository on GitLab for this assignment called "pa2". Some test files will be included in a folder within "pa2", and the assembly code for these will be created through Professor Miller's RISC-V assembler website.

There's new framework to implement, so we can't simply copy the code from our previous assignment. This design document explains how our code works with the new framework.

Debugging:

While debugging, if we decide to use printf() statements, surround them with preprocessor statements.

```
#ifdef DEBUG
printf("This sentence should tell you something useful.\r\n");
#endif
```

This way, it's faster to comment out many printf() statements and put them back in.
To turn on debugging, simply put this statement at the top of the code:
`#define DEBUG`

To create new lines, the UNIX timeshare requires that you use "\r\n" at the end of your statement, not just "\n".

We will visualize the pipeline flow by printing out the status of each stage as it occurs. This should help with pinpointing where bugs occur.

Unit Testing:

- We will use test suites to organize unit testing for each stage separately
- Unit tests should not rely on each other and must be completely isolated
    - Running the tests in a random order should always return the same results!
- Overall results will be displayed visually as PASS or FAIL, and the user will be notified of which specific tests failed
- We will not be using any external assertion library for testing
- As mentioned earlier, the complete test suite can be run with sim -u
- "Simulating the simulator" will be difficult to integrate into unit testing, so we'll have to make sure each test doesn't rely on the pipeline
- Unit testing will not be able test the pipeline flow, so we'll have to include integration testing as well
- We have to consider making each individual unit test efficient, but also have as much coverage as possible by creating numerous tests
    - Unit tests will serve as a reliable testing foundation and give us confidence that our lower-level code is working as planned (i.e. decoding)
- We should run the test suite frequently to ensure that any code refactoring hasn't broken anything

---

Code:

STALLING:

To stall any stage, have the preceding stage reload the same values into the stage register rather than loading new values.

FORWARDING:
We must support three types of forwarding: MX, WX, and WM. Our general implementation will be the same for each type.

NOP:
All stages should check for a NOP flag. If one is detected, the stage shouldn't do anything.

FETCH stage:

To implement branch prediction, we will use a Branch Target Buffer (BTB) array. The BTB will be a global array of branch structs which have fields for the tag, target, and address. Each element in the array will be indexed by the low order bits of the address.

- Address: Lowest 5 bits of address, used to index branch within BTB.
- Tag: The remaining portion of the address (59 bits), used to find match.
- Target: The destination in memory that the branch points to.

Branch prediction starts here.

First, we read the instruction from memory during the current (fetch) stage.

To do our branch prediction, we will have a Branch Target Buffer with 32 entries. Using a mask, we extract the low-order bits. If the low-order bits match, we then shall compare the tag of the address to the BTB. If those match too, then we will branch to our target, and start fetching instructions from our target.

DECODE stage:

Decode the instruction. We can simply reuse code from last time for this stage.

Every type of instruction is decoded by first reading the last 7 bits of the instruction, called the *opcode.*

We will be considering 5 types of instructions: R-type, I-type, S-type, SB-type, and U-type instructions.

While reading the opcode, we find out what type of instruction it is.

R-type instructions:

Description: Arithmetic instructions

Instruction Format:

{funct7}{rs2}{rs1}{funct3}{rd}{opcode}
The opcode has 2 possible values:

- 0x33: Arithmetic
- 0x3B: Wide arithmetic

rs1: argument 1 register
rs2: argument 2 register
rd: result register
funct3: primary function determiner
funct7: secondary function determiner

Functions:

Arithmetic:

- add: funct3 = 0 and funct7 = 0
- sub: funct3 = 0 and funct7 = 0x20
- sll: funct3 = 0x1
- slt: funct3 = 0x2
- sltu: funct3 = 0x3
- xor: funct3 = 0x4
- srl: funct3 = 0x5, funct7 = 0
- sra: funct3 = 0x5, funct7 = 0x20
- or: funct3 = 0x6
- and: funct3 = 0x7

funct7 = 0x1 for all of the below:

- mul: funct3 = 0
- mulh: funct3 = x1
- mulhsu funct3 = x2
- mulhu: funct3 = x3
- div: funct3 = x4
- divu: funct3 = x5
- rem: funct3 = x6
- remu: funct3 = x7

Wide Arithmetic:

- addw: funct3 = 0, funct7 = 0
- subw: funct3 = 0, funct7 = x20

- sllw: funct3 = x1,
- srlw: funct3 = x5, funct7 = 0
- sraw: funct3 = x5, funct7 = x20

funct7 = 0x1 for all of the below:

- mulw: funct3 = 0
- divw: funct3 = x4
- divuw: funct3 = x5
- remw: funct3 = x6
- remuw:funct3 = x7

I-type instructions:

Description: Loads, immediate arithmetic, and JALR.

Instruction Format:

{imm[11:0]}{rs1}{funct3}{rd}{opcode}
Opcode is not fixed here. There are 4 different opcodes this could be:

- 0x13: Immediate Arithmetic.
- 0x03: Loads
- 0x1B: Wide Immediate Arithmetic
- 0x67: JALR function.

rd: target register to write to
rs1: argument register.
imm: what to add to rs1. Must be sign extended.
funct3: determines what type of load or arithmetic executed.

Functions:

Immediate Arithmetic:

- addi: funct3 = 0
- slli: funct3 = 1
- slti: funct3 = 2
- sltiu: funct3 = 3
- xori funct3 = 4
- srli funct3 = 5, imm[11:5] = 0b0000000
- srai funct3 = 5, imm[11:5] = 0b0100000
- ori funct3 = 6,
- andi funct3 = 7,

Loads:

- lb: funct3 = 0
- lh: funct3 = 1
- lw: funct3 = 2

- ld: funct3 = 3
- lbu: funct3 = 4
- lhu: funct3 = 5
- lwu funct3 = 6

Wide Immediate Arithmetic:

- addiw funct3 = 0
- slliw funct3 = 1
- srliw funct3 = 2
- sraiw funct3 = 3

JALR function:

Simply returns the PC to rs1 + imm, and writes old pc + 4 to rd.

S-type instructions:

Description: Stores

Instruction format:

{imm[11:5]}{rs2}{rs1}{funct3}{imm[4:0]}{opcode}

The opcode is 7 bits. It's always 0x23 for S-type instructions.
funct3: determines how big of a byte to store.
imm is 12 bits that must be sign extended to 32 bits. Determines the offset.
target address to store to: rs1 + imm
register that contains value to store to memory: rs2

Functions:

- sb: funct3 = 0x00. store 1 byte
- sh: funct3 = 0x01. store 2 bytes
- sw: funct3 = 0x02. store 4 bytes
- sd: funct3 = 0x03. store 8 bytes.

SB-Type instructions:

Description: Conditional branch format

Instruction Format:

{imm[12]}{imm[10:5]}{rs2}{rs1}{funct3}{imm[4:1]}{imm[11]}{opcode}

opcode: 0b11000.
funct3: 3 bits. Determines
rs1: 5 bits.
rs2: 5 bits.
imm: determines offset for branching.
The new pc is the old pc plus offset.

Functions:

- beq: funct3 = 0x00. Branches if rs1 == rs2
- bne: funct3 = 0x01. Branches if rs1 != rs2
- blt: funct3 = 0x04. Branches if rs1 < rs2
- bge: funct3 = 0x05. Branches if rs1 >= rs2
- bltu: funct3 = 0x06. Branches if rs1 < rs2, unsigned comparison.
- bgeu: funct3 = 0x07. Branches if rs1 >= rs2, unsigned comparison.

U-Type instructions:

Description: Upper immediate format

Instruction Format:

{imm[31:12]}{rd}{opcode}
There are 3 possible opcodes for U-type instructions. We will hard-code them all.

rd is the target register.

Functions:

- lui : opcode = 0x17: load immediate into rd.
- auipc: opcode = 0x37: Add upper immediate to PC, save new pc to rd.
- jal: opcode = 0x6F: Jump and link: save PC + 4 to rd, set new pc to pc + imm.

Implementation:

This stage is implemented with the function:

```
void stage_decode (struct stage_reg_x *new_x_reg);

struct stage_reg_x {
    uint64_t    new_pc;
    uint32_t    instruction;
};
```

We need to write things to new_x_reg to pass it to the execute stage next time.
We can grab things written from the previous fetch stage from the variable:

```
extern const struct stage_reg_d *current_stage_d_register;
```

We need to find a way to stall the decode instruction because the execute stage will inevitably take longer to complete than the decode stage at some point.

To implement stalling:

Check the previous instruction by checking the execute register. If the target register there is one of the source registers here, stall the decode instruction.

To pass a function to the execute stage, we shall have an enumerated type for every instruction. ex:

```
enum instr_type {
        addi =0, addiw=1, ….. beq=0x111;
}
```

This way, the execute stage doesn't have to decode again and can exclusively perform the operation on the fetched values.

If the decode stage needs values that aren't available yet, we must stall the decode stage. To check if it needs values, check if the source registers to both the memory destination and the execute destination.

To stall, send the value back into the decode function, and send NOP into the execute stage. The decode stage shall pass the 2 argument values and the destination register to

EXECUTE stage:

Execute the instruction in question by identifying the ALU operation and using the received values from the decode stage as operands. This way we won't have to decode again to actually execute the instruction.

How are we going to implement data forwarding? We shall do it with our data registers.

Load/store instructions calculate the address to read/write to in this stage.
Arithmetic instructions do the math in this stage.
Branch instructions calculate the address to branch to in this stage, and if we are branching. If there's a misprediction, we must flush the other instructions.

Assuming the memory stage always takes one cycle, we never have to stall the execute stage.

To forward the data to the decode stage, put the calculated value into new_m_reg->mx_forwarding_variable.

Branch prediction starts here.

If the type of instruction is a branch, we will execute this extra step.
If the branch target in the BTB is the same as the calculated address, we keep going.
If the branch target in the BTB is not the same as the calculated address, we must flush the instructions that we loaded.

To flush instructions, we will have the execute stage pass a flush flag to new_m_reg.
Execute and decode will check for this flag; when they see this flag has been set, they will set their instructions to NOP.

MEMORY stage:

Read from or write to memory (if the instruction requires it).

These functions use the memory stage:
Load instructions read from memory in this stage, using memory_read().
Store instructions write to memory in this stage, using memory_write().
Branch instructions set the program counter in this stage, using set_pc().

We do forwarding from M to X here. We will pass this value to new_w_reg->wx_forwarding_variable.

WRITEBACK stage:

Writing values to registers must occur exclusively in this stage.

Example: in load instructions, once the data has been loaded from memory, you write the loaded contents back to the destination register.

For all instructions, register rd is written here. We shall use the register_write() function to accomplish that.

Example: register_write(rd, value);

Note: the writeback stage does not pass values to stage registers, as it lies at the end of the pipeline.

Deliverables:

1. Makefile
2. README
3. Design Document
4. Code

Code Files:

```
riscv_pipeline_registers.h // lets the other files access current
registers.
riscv_sim_pipeline_framework.c // implements the whole framework
riscv_sim_pipeline_framework.h
riscv_sim_pipeline_stage_fetch.c // implements stage_fetch()
riscv_sim_pipeline_stage_fetch.h
riscv_sim_pipeline_stage_decode.c // implements stage_decode()
riscv_sim_pipeline_stage_decode.h
riscv_sim_pipeline_stage_execute.c // implements stage_execute()
riscv_sim_pipeline_stage_execute.h
riscv_sim_pipeline_stage_memory.c // implements stage_memory()
riscv_sim_pipeline_stage_memory.h
riscv_sim_pipeline_stage_writeback.c // implements stage_writeback()
riscv_sim_pipeline_stage_writeback.h
global_variables.h
```

References:

Assignment 2 Handout

RISC-V Specifications Version 2.2

Professor Miller's website:
http://elm-vm-1.soe.ucsc.edu/cmpe110/riscv-assemble.html

CMPE110 Architecture Book

Piazza class page