Introduction to Computer Architecture
Project 3

Pipelined MIPS CPU Simulator

Hyungmin Cho
Department of Software
Sungkyunkwan University

Project 3 Overview

- In Project 3, you’ll extend the instruction simulator from Project 2 to simulate a pipelined MIPS processor

- The basic rules (submission rule, etc…) are the same as Project 2, but please ask TAs if anything is unclear.

Subset of MIPS Instructions to Support in Proj3

- Same as Proj2:
  - Arithmetic/logical: **`add, sub, and, or, slt,`**
  - Arithmetic/logical with immediate: **`addi, andi, ori, slti, lui`**
  - Memory access: **`lw, sw`**
  - Control transfer: **`beq, bne, j`**
  - No-operation: **`nop (0x00000000)`**

- **Ignore unknown instructions `(e.g., 0xFFFFFFFF)`**
  - Unlike proj2, **do not stop** or print something when you encounter an unknown instruction.
  - Instead, if you encounter the instruction 0xFFFFFFFF (*i.e.*, the default memory value), make sure you **do not write anything into the registers or memory**.

Specification of the Pipelined MIPS Processor (1)

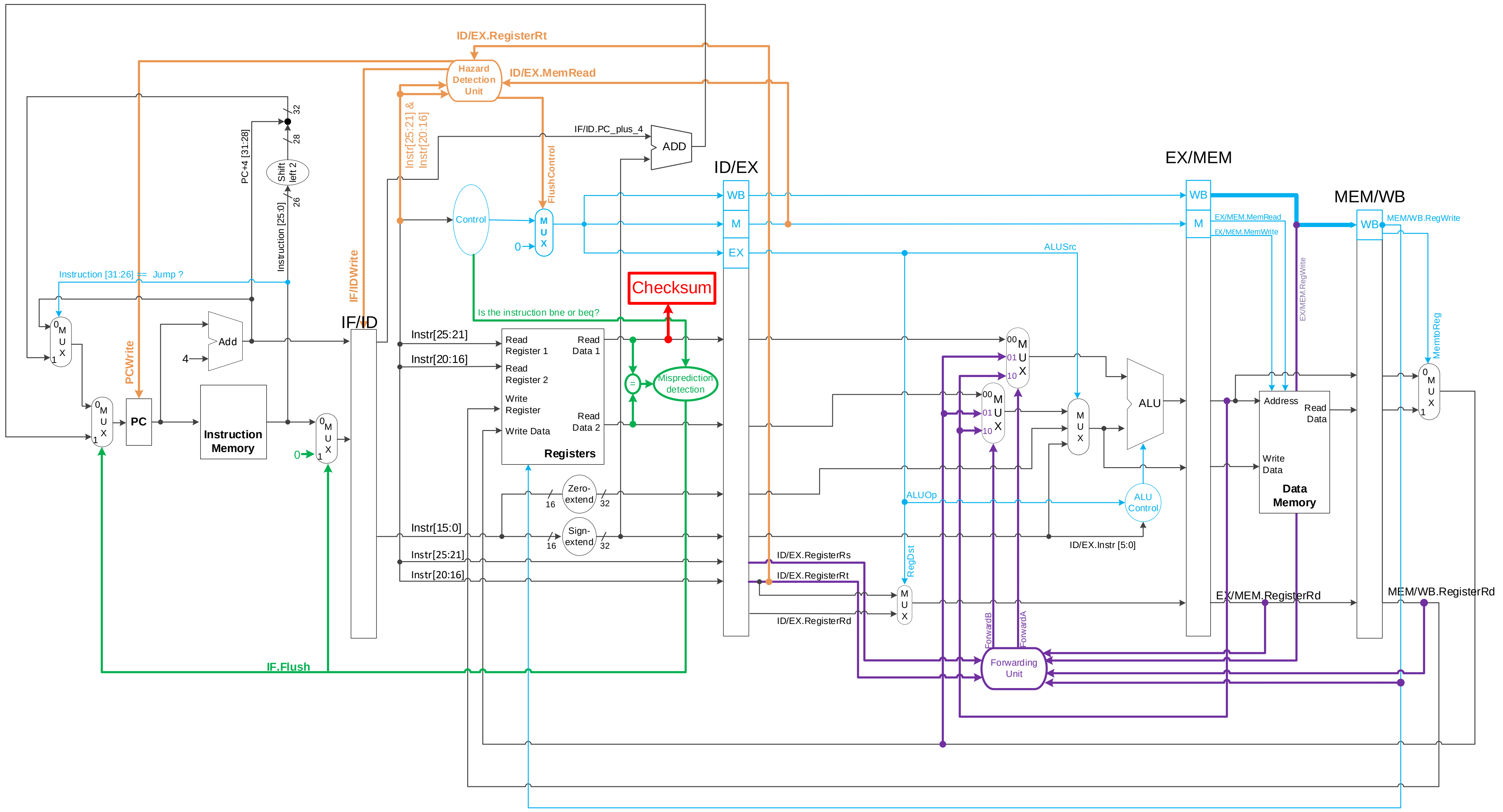
- 5-stage MIPS processor as we studied in the class

- Data Hazard
  - Register bypass
  - EX / MEM hazards are handled through forwarding
  - Load-use hazard add one bubble cycle
  - If a branch instruction has a data hazard, it is also handled through forwarding!

Specification of the Pipelined MIPS Processor (2)

- Control hazard
  - No delayed branch. That means there is no “branch delay slot”. If a branch is turned out to be a taken branch, the instruction after the branch instruction (*i.e.*, PC+4) should not be executed.
    - If the PC+4 instruction is already inserted to the pipeline by the branch prediction, it should be canceled.
    - This is the mechanism we explained in the lecture, but I’m writing it here again to make sure not to be confused with the “branch delay slot” part in the textbook…
  - “Always not taken” branch prediction
  - The actual outcome of the branch is resolved at the ID stage
    - On a misprediction, flush the PC+4 instruction in the IF stage and set PC to the jump target address.
  - Jump instruction is handled in the IF stage (no added bubble/flush cycle for Jump).
    - See the figure in Page 5 to see how a jump is handled.

The Processor Overview



The Processor Overview

- The figure in the previous page describes the processor structure that includes the following:
  - Default pipelined datapath and control
  - Register bypass (not drawn, but as we discussed in the lecture, it is implicitly assumed)
  - Forwarding logic (Purple)
  - Load-use hazard detection and bubble insertion (Orange)
  - Always-not-taken Branch prediction and misprediction detector (Green)

- However, it misses one critical point:
  - Forwarding logic for the branch misprediction detector!
  - Figure out how to implement such a functionality (tested by `proj3_6.bin and proj3_7.bin`)

Register Read Checksum

- A pipelined microprocessor should have the same results (register, memory) as a single-cycle processor. This means… it can be tricky to "verify" if you have correctly implemented a pipelined microprocessor.

- You need to add an extra "**Checksum**" module to check if the pipeline worked as required.
  - The checksum computes the following **on every cycle**
    - `Checksum = (Checksum <<1 | Checksum >>31)  XOR Register_read_data_1`
    - `Checksum` is a 32-bit value initialized to 0x00000000 at beginning.
    - That is, on every cycle, perform "barrel shift to the left by 1 bit" on the checksum and perform XOR with the `$rs` register read value.
    - `Register_read_data_1` is the output of the register file's first read data (IF/ID.instruction[25:21]), after the register bypass logic. It is NOT the forwarded value at the EX stage!
    - The checksum should be calculated on every cycle, even if the ID stage is processing an instruction that does not use the `$rs` register or processing a bubble.

Simulator Program Behavior

1. Similar to proj2, the input file name is given as the first command-line argument.

2. Your program reads the file and load the binary instructions to address `0x00000000` of the instruction memory.

3. Your program simulates each instruction one-by-one, up to ***N*** **cycles**.
   - ***N*** is given as the second command-line argument.
   - For Proj 3, ***N*** is the number of cycles, not the number of instructions!

4. ~~If the simulator reads an unsupported instruction before~~ ~~***N***~~ ~~instructions, print “`unknown instruction`” and exit.~~ ← Do not stop at an unknown instruction

5. Before the exit, depending on the third command-line argument, print the final status of the system.

Output Options (1) – Little bit different from Proj2

- If the third command-line argument is “reg”, print the register values in the following format

```
# ./mips-sim data.bin 10 reg
Checksum: 0x0000001e
$0: 0x00000000
$1: 0x00000000
$2: 0x0000000a
$3: 0x00000014
$4: 0x10000004
$5: 0x1000002c
$6: 0x00000000
$7: 0x00000000
$8: 0x00000000
$9: 0x00000000
$10: 0x00000000
$11: 0x00000000
$12: 0x00000000
$13: 0x00000000
$14: 0x00000000
$15: 0x00000000
$16: 0x00000000
$17: 0x00000000
$18: 0x00000000
$19: 0x00000000
$20: 0x00000000
$21: 0x00000000
$22: 0x00000000
$23: 0x00000000
$24: 0x00000000
$25: 0x00000000
$26: 0x00000000
$27: 0x00000000
$28: 0x00000000
$29: 0x0fffffd0
$30: 0x0fffffd0
$31: 0x00000370
PC: 0x0000037c
#
```

- Print the current value of **Checksum** at cycle *N*

- Print the register values in the register file at the current cycle, ***after*** the writeback stage has updated the target register.

- The PC value should be pointing to the address of the fetched instruction in the IF stage

Output Options (2) – Same as Proj2

- If the third command-line argument is “mem”, the program expects the fourth and fifth argument .
  - 4th argument : Starting address to print from the data memory (in hexadecimal)
  - 5th argument : Number of 4-byte words to print (in decimal)

```
# ./mips-sim data.bin 10 mem 0x10000010 6
0x00000001
0x00000012
0x00000123
0x00001234
0xFFFFFFFF
0xFFFFFFFF
#
```

Data memory value at 0x10000010
Data memory value at 0x10000014
Data memory value at 0x10000018
Data memory value at 0x1000001C
Data memory value at 0x10000020
Data memory value at 0x10000024

Output Options (3) – Same as Proj2

- If there is no third command-line argument or an incorrect argument is given, no need to print anything.

```
# ./mips-sim data.bin 1000
#
```

Test Sample (1)

- `~swe3005/2020f/proj3/proj3_1.bin`

- "proj3_1.bin" file represents the following assembly code.

```
addi $2, $0, 0x1234
lui  $3, 0x8765
addi $4, $0, -1
slt  $5, $2, 1234
ori  $6, $3, 0x4321
andi $7, $4, 0xFFFF
addi $8, $4, -4
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_1.bin 11 reg
```

```
Checksum: 0xeca48d10
$0: 0x00000000
$1: 0x00000000
$2: 0x00001234
$3: 0x87650000
$4: 0xffffffff
$5: 0x00000000
$6: 0x87654321
$7: 0x0000ffff
$8: 0xfffffffb
$9: 0x00000000
...
$31: 0x00000000
PC: 0x00000028
```

- proj3_1.bin has no register forwarding, no branch, and no load-use data hazard
- The 5-stage pipelined execution of these 7 instructions would take 11 cycles (7 + 5 – 1).

Test Sample (2)

- `~swe3005/2020f/proj3/proj3_2.bin`

- "proj3_2.bin" file represents the following assembly code.

```
addi $2, $0, 0x4321
lui  $3, 0x8765
ori  $4, $0, 0xFFFF
ori  $5, $2, 0xABCD
slti $6, $3, 0x4000
addi $7, $4, 0x1
nop
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_2.bin 11 reg
```

```
Checksum: 0xecbf37a0
$0: 0x00000000
$1: 0x00000000
$2: 0x00004321
$3: 0x87650000
$4: 0x0000ffff
$5: 0x0000ebed
$6: 0x00000001
$7: 0x00010000
$8: 0x00000000
$9: 0x00000000
...
$31: 0x00000000
PC: 0x00000028
```

- proj3_2.bin has register bypassing at cycle 5
- At cycle 5, "`ori  $5, $2, 0xABCD`" instruction reaches the ID stage to read $2, which is being written by the "`addi $2, $0, 0x4321`" instruction. Make sure the `ori` instruction reads the updated value!

Test Sample (3)

- `~swe3005/2020f/proj3/proj3_3.bin`

- "proj3_3.bin" file represents the following assembly code.

```
addi $1, $0, 100
addi $2, $1, -50
nop
nop
nop
addi $3, $2, 200
nop
addi $4, $3, 300
nop
nop
addi $5, $4, 400
nop
nop
lui $3, 0x1000
addi $3, $3, 10
addi $4, $0, 0xBEEF
sw $4, 14($3)
nop
nop
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_3.bin 23 reg
```

```
Checksum: 0x00234d02
$0: 0x00000000
$1: 0x00000064
$2: 0x00000032
$3: 0x1000000a
$4: 0xffffbeef
$5: 0x000003b6
$8: 0x00000000
$9: 0x00000000
...
$31: 0x00000000
PC: 0x00000058
```

- proj3_3.bin has register forwarding at cycle 4, 10, 17, 19

Test Sample (4)

- `~swe3005/2020f/proj3/proj3_4.bin`

- "proj3_4.bin" file represents the following assembly code.

```
lui $3, 0x1000
addi $3, $3, 40
addi $4, $0, 0x1398
add  $5, $4, $3
sw $4, 0($3)
sw $5, 4($3)
nop
nop
lw $5, 0($3)
addi $6, $5, -123
nop
lw $7, 4($3)
nop
addi $8, $7, -456
nop
nop
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_4.bin 22 reg
```

```
Checksum: 0x00efb1d0
$0: 0x00000000
$1: 0x00000000
$2: 0x00000000
$3: 0x10000028
$4: 0x00001398
$5: 0x00001398
$6: 0x0000131d
$7: 0x100013c0
$8: 0x100011f8
$9: 0x00000000
...
$31: 0x00000000
PC: 0x00000050
```

- proj3_4.bin has load-use data hazard when the instruction that uses the loaded value (addi) reaches the ID stage at cycle 11.

Test Sample (5)

- `~swe3005/2020f/proj3/proj3_5.bin`

```
./mips-sim ~swe3005/2020f/proj3/proj3_5.bin 22 reg
```

```
Checksum: 0x01818300
$0: 0x00000000
...
$9: 0x00000000
$10: 0x10000008
$11: 0x00003090
$12: 0x00003091
$13: 0x00003091
$14: 0x00003000
$15: 0x00000000
...
$31: 0x00000000
PC: 0x0000003c
```

- “proj3_5.bin” file represents the following assembly code.

```
    lui $10, 0x1000
    ori $11, $0, 0x3090
    sw $11, 4($10)
    addi $12, $11, 1
    sw $12, 8($10)
L1: addi $10, $10, 4
    lw $13, 0x0($10)
    nop
    nop
    bne $12, $13, L1
    andi $14, $13, 0xFF00
```

- proj3_5.bin tests branch prediction

Test Sample (6)

- `~swe3005/2020f/proj3/proj3_6.bin`

- “proj3_6.bin” file represents the following assembly code.

```
    lui $10, 0x1000
    ori $11, $0, 0x3090
    sw $11, 4($10)
    addi $12, $11, 1
    sw $12, 8($10)
L1: addi $10, $10, 4
    lw $13, 0x0($10)
    nop
    bne $12, $13, L1
    andi $14, $13, 0xFF00
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_6.bin 21 reg
```

```
Checksum: 0x018ebc50
$0: 0x00000000
...
$9: 0x00000000
$10: 0x10000008
$11: 0x00003090
$12: 0x00003091
$13: 0x00003091
$14: 0x00003000
$15: 0x00000000
...
$31: 0x00000000
PC: 0x0000003c
```

- proj3_6.bin tests branch prediction + forwarding

Test Sample (7)

- `~swe3005/2020f/proj3/proj3_7.bin`

- "proj3_7.bin" file represents the following assembly code.

```
    lui $10, 0x1000
    ori $11, $0, 0x3090
    sw $11, 4($10)
    addi $12, $11, 1
    sw $12, 8($10)
L1: addi $10, $10, 4
    lw $13, 0x0($10)
    bne $12, $13, L1
    andi $14, $13, 0xFF00
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_7.bin 21 reg
```

```
Checksum: 0x029fe4d0
$0: 0x00000000
...
$9: 0x00000000
$10: 0x10000008
$11: 0x00003090
$12: 0x00003091
$13: 0x00003091
$14: 0x00003000
$15: 0x00000000
...
$31: 0x00000000
PC: 0x00000038
```

- proj3_7.bin tests branch instruction + load use data hazard
  - Carefully examine the output of the reference implementation at cycles "9, 10, 11" and "14,15,16"

Test Sample (8)

- `~swe3005/2020f/proj3/proj3_8.bin`
- “proj3_8.bin” file represents the following assembly code.

```
    addi $2, $0, 1000
    addi $3, $0, 2000
    addi $4, $0, 3000
    addi $5, $0, 4000
    j L2
    ori $10, $0, 0xDEAD
    ori $11, $0, 0xDEAD
L1: addi $2, $2, 1
    addi $3, $3, 2
L2: addi $4, $4, 3
    addi $5, $5, 4
    j L1
    ori $12, $0, 0xDEAD
    ori $13, $0, 0xDEAD
    nop
    nop
```

```
./mips-sim ~swe3005/2020f/proj3/proj3_8.bin 40 reg
```

```
Checksum: 0x94d3e64e
$0: 0x00000000
$1: 0x00000000
$2: 0x000003ee
$3: 0x000007dc
$4: 0x00000bcd
$5: 0x00000fb8
$6: 0x00000000
...
$31: 0x00000000
PC: 0x00000024
```

- proj3_8.bin tests jump instructions (infinite loop)

Reference Implementation (1)

- We provide a reference implementation (without source code) in the following location.
  - `~swe3005/2020f/proj3/mips-sim`

Reference Implementation (2)

- The reference implementation support the following feature to help your implementation, but the students **do not need to implement** this feature
  - In addition to ***reg*** and ***mem*** options, the reference implementation supports "***pipe***" option to show the detailed status of the pipeline at cycle *N*.

```
proj3_1.bin
```

```
addi $2, $0, 0x1234
lui  $3, 0x8765
addi $4, $0, -1
slt  $5, $2, 1234
ori  $6, $3, 0x4321
andi $7, $4, 0xFFFF
sub  $8, $4, 4
```

```
~swe3005/2020f/proj3/mips-sim ~swe3005/2020f/proj3/proj3_1.bin 4 pipe
```

```
Checksum: 0x00000000 = (0x00000000<<1|0x00000000>>31) ^ 0x00000000
IF: slti $5, $2, 1234, Current PC:0x0000000c Next PC:0x00000010
ID: addi $4, $0, -1, imm[15:0]: 0xffff rs:0=0x00000000 rt:4=0x00000000
EX: lui $3, -30875, ALU_input_1:0x00000000 ALU_input_2:0xffff8765 ALUResult:0x87650000 ForwardA:0 ForwardB:0 ALUSrc:2 RegDst:0
MEM: addi $2, $0, 4660, Address:0x00001234 memory_write_data:0x00000000 memory_read_data: 0x00000000 MemRead:0 MemWrite:0
WB: sll $0, $0, 0, ALUresult: 0x00000000 memory_read_data: 0x00000000 WriteRegister:0x00000000 MemtoReg:0 RegWrite:1
```

Reference Implementation (3)

- If the current cycle has a branch misprediction or load-use data hazard, the “pipe” option will display the detailed info.

```
proj3_5.bin
```

```
 lui $10, 0x1000
    ori $11, $0, 0x3090
    sw $11, 4($10)
    addi $12, $11, 1
    sw $12, 8($10)
L1: addi $10, $10, 4
    lw $13, 0x0($10)
    nop
    nop
    bne $12, $13, L1
    andi $14, $13, 0xFF00
```

```
~swe3005/2020f/proj3/mips-sim ~swe3005/2020f/proj3/proj3_5.bin 11 pipe
```

```
Checksum: 0x80003092 = (0xc0000001<<1|0xc0000001>>31) ^ 0x00003091
IF: andi $14, $13, -256, Current PC:0x00000028 Next PC:0x00000014
ID: bne $12, $13, -5, imm[15:0]: 0xfffb rs:12=0x00003091 rt:13=0x00003090
EX: sll $0, $0, 0, ALU_input_1:0x00000000 ALU_input_2:0x00000000 ALUResult:0x10000004 ForwardA:0 ForwardB:0 ALUSrc:0 RegDst:1
MEM: sll $0, $0, 0, Address:0x10000004 memory_write_data:0x00000000 memory_read_data: 0x00000000 MemRead:0 MemWrite:0
WB: lw $13, 0($10), ALUresult: 0x10000004 memory_read_data: 0x00003090 WriteRegister:0x0000000d MemtoReg:1 RegWrite:1
Branch misprediction detected at the ID stage
  Target branch address: 00000014
  Flushed instruction in IF stage was andi $14, $13, -256
```

Reference Implementation (4)

- The reference implementation uses an arbitrary value if the control is "don't care (***X***)".
  - For example, the `RegDst` value for a store instruction is don't care, but the reference implementation assigns "0".
  - You don't need to match all outputs of the reference implementation shown with the "pipe" option as long as your implementation's output matches the registers, memory, and Checksum values of the reference.

Project Environment

- We will use the department's ~~In-~~Ui-Ye-Ji cluster
  - ~~`swin.skku.edu`~~
  - `swui.skku.edu`
  - `swye.skku.edu`
  - `swji.skku.edu`
  - ssh port: 1398

- You'll need a similar Makefile as proj2
  - Same executable file name (i.e., **`mips-sim`**)

Submission

- Clear the build directory
  - Do not leave any executable or object file in the submission

- Use `submit` program
  - `~swe3005/bin/submit project_id path_to_submit`
  - If you want to submit the current directory…
    - `~swe3005/bin/submit` **`proj3 .`**

```
Submitted Files for proj3:
File Name                                File Size       Time
--------------------------------------------------------------------------------
proj3-2020123456-Sep.05.17.22.388048074        268490                 Thu Sep  5 17:22:49 2020
```

- Verify the submission
  - `~swe3005/bin/check-submission` **`proj3`**

Project 3 Due Date

- 2020 Dec 20th, 23:59:59
- No late submission