4190.308: Computer Architecture Lab. 4

Jaehoon Shim
Ikjoon Son
Seongyeop Jeong
(snucsl.ta@gmail.com)

Systems Software & Architecture Lab.

Seoul National University

Fall 2021

SNURISC-SEQ

SNURISC-SEQ

Get the instruction indicated by pc

Get values required for instruction

Perform computation

Access data memory

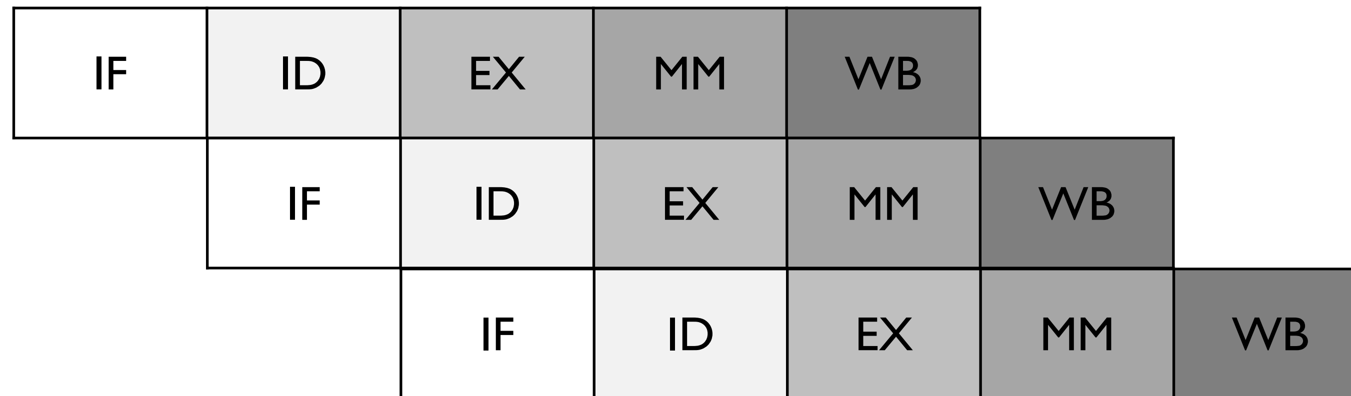
Write back data to register file

The PyRISC Project
SNURISC-SEQ Diagram
https://github.com/snu-csl/pyrisc

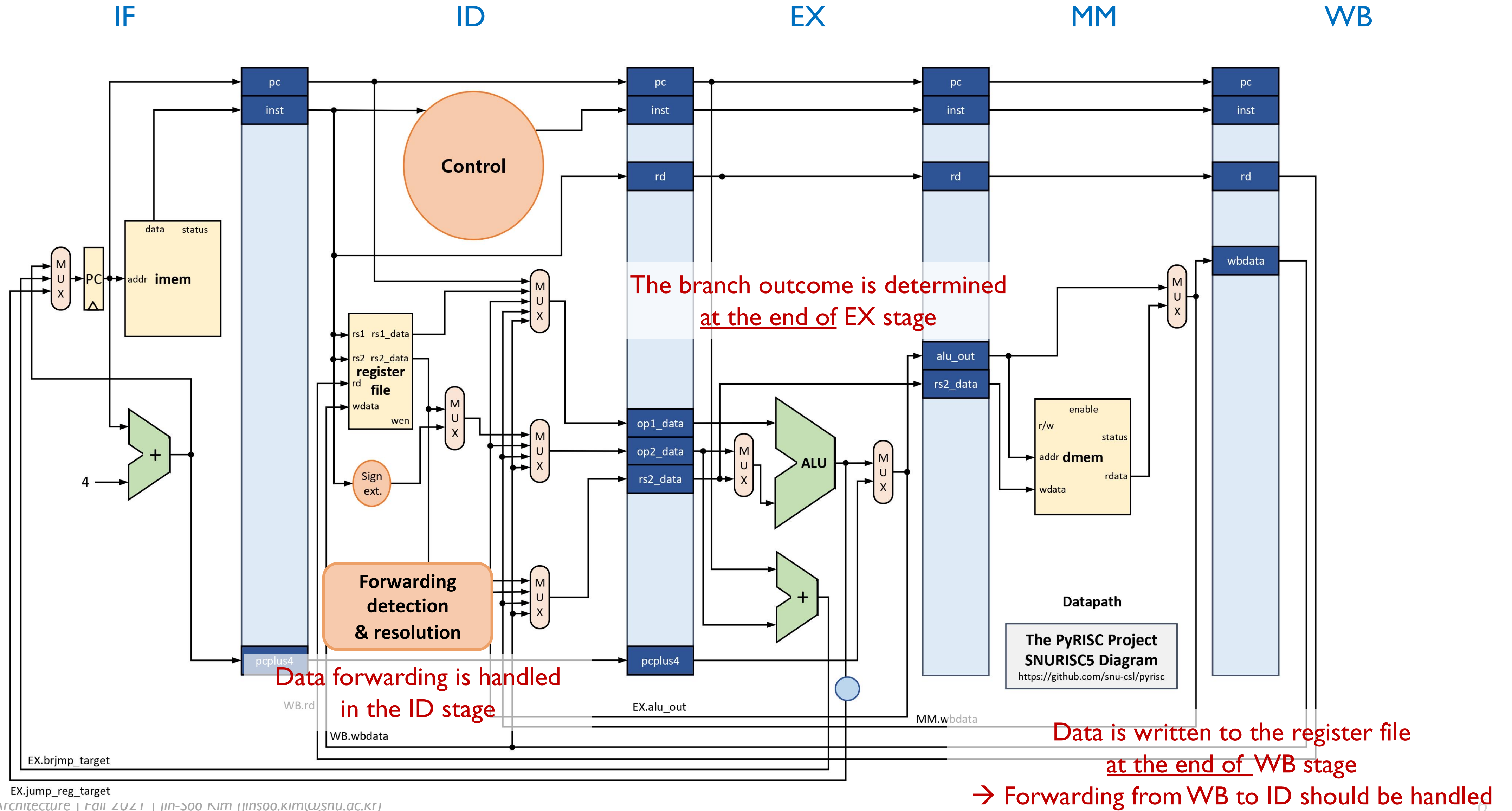
SNURISC5

SNURISC5

- A 5-stage pipelined RISC-V Simulator
- It consists of
  - IF: Instruction fetch
  - ID: Instruction decode
  - EX: Execute
  - MM: Memory access
  - WB: Writeback



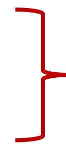
SNURISC5



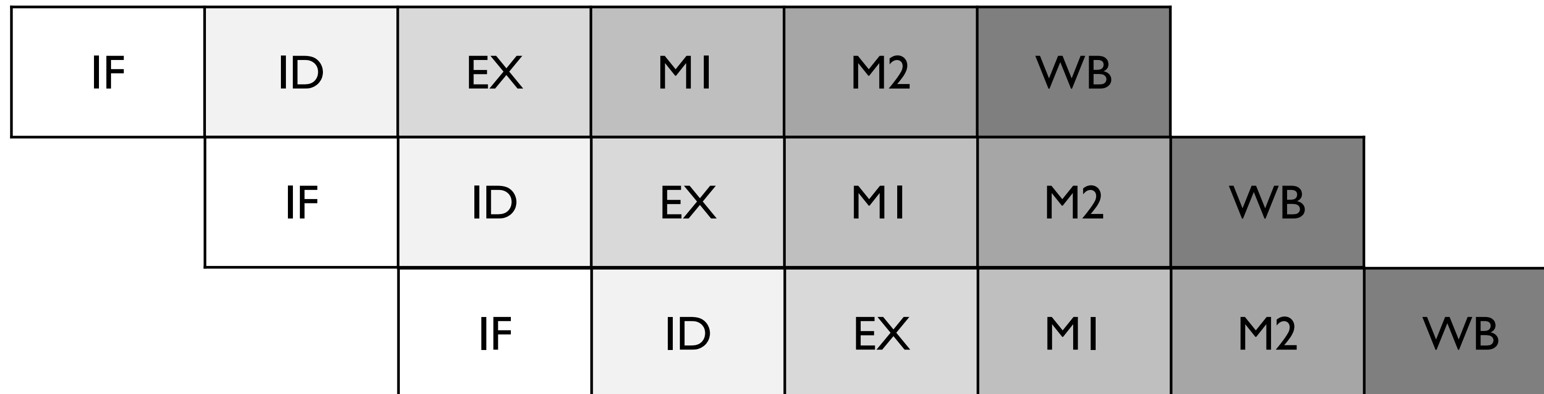
SNURISC6

SNURISC6

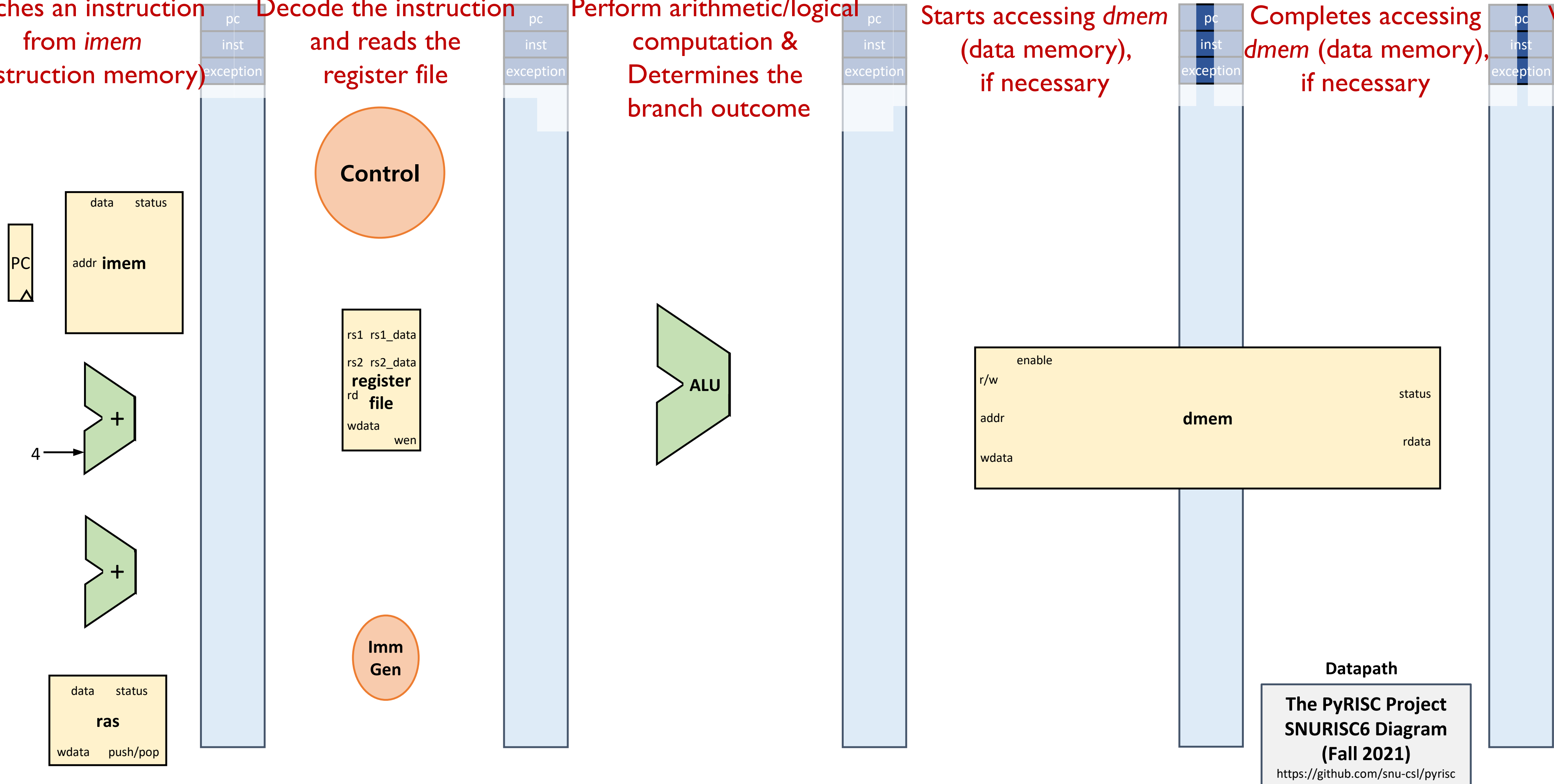
- A 6-stage pipelined RISC-V Simulator
- It consists of
  - IF
  - ID
  - EX
  - M1: Starts accessing dmem
  - M2: Completes accessing dmem
  - WB



MM stage in traditional 5-stage pipeline is divided into two stages



SNURISC6



SNURISC6

- Overall simulator architecture
  - snurisc6.py: It parses arguments from the user and controls the overall simulation
  - program.py: It loads the contents of the input RISC-V executable file to *imem*
  - pipe.py: It controls the actual execution of the simulation
  - stage.py: It contains the datapath information for each stage and the control logic
  - components.py: It has various hardware components such as RegisterFile, Register, Memory, ALU, Adder, and RAS
  - isa.py: It has definition of each instructions and decoding logic for RISC-V instruction set
  - consts.py: It defines various constants used throughout the simulator

SNURISC6

- class Pipe (in pipe.py)

```
def set_stages(cpu, stages):
    Pipe.cpu = cpu
    Pipe.stages = stages
    Pipe.IF = stages[S_IF]
    Pipe.ID = stages[S_ID]
    Pipe.EX = stages[S_EX]
    Pipe.M1 = stages[S_M1]
    Pipe.M2 = stages[S_M2]
    Pipe.WB = stages[S_WB]
```

Each points to the corresponding objects of IF, ID, EX, M1, M2, and WB classes

```
def run(entry_point):
    IF.reg_pc = entry_point
    while True:
        Pipe.WB.compute()
        Pipe.M2.compute()
        Pipe.M1.compute()
        Pipe.EX.compute()
        Pipe.ID.compute()
        Pipe.IF.compute()
        # Update states
        Pipe.IF.update()
        Pipe.ID.update()
        Pipe.EX.update()
        Pipe.M1.update()
        Pipe.M2.update()
        ok = Pipe.WB.update()

        if not ok:
            break
```

Reverse order due to dependence of hazard/forwarding detection

Manipulation of signals using some combinational logic performed inside of the stage

Contents of the pipeline registers are updated

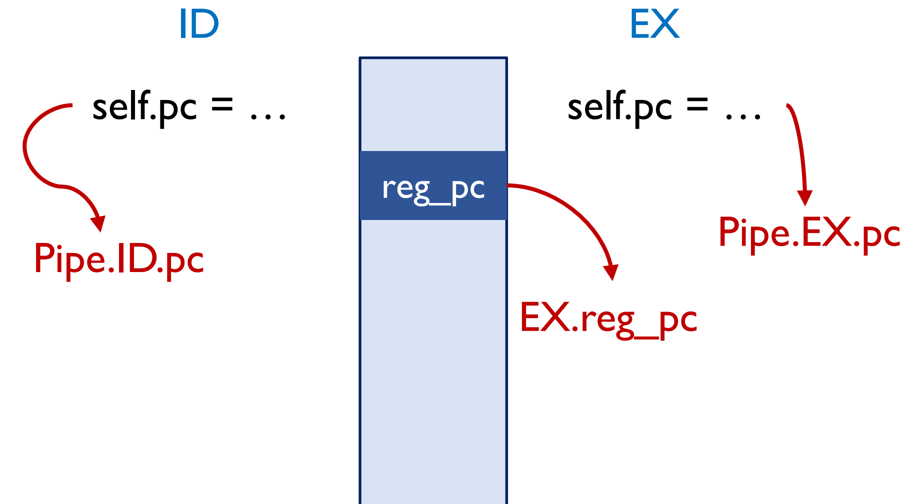
SNURISC6

- Naming convention
  - Pipeline registers
    - Implemented as class variables
      → referenced as [class name].[variable name]
    - Prefix 'reg_' is added

    e.g., EX.*reg_pc*: pipeline register '*reg_pc*' between ID and EX stage

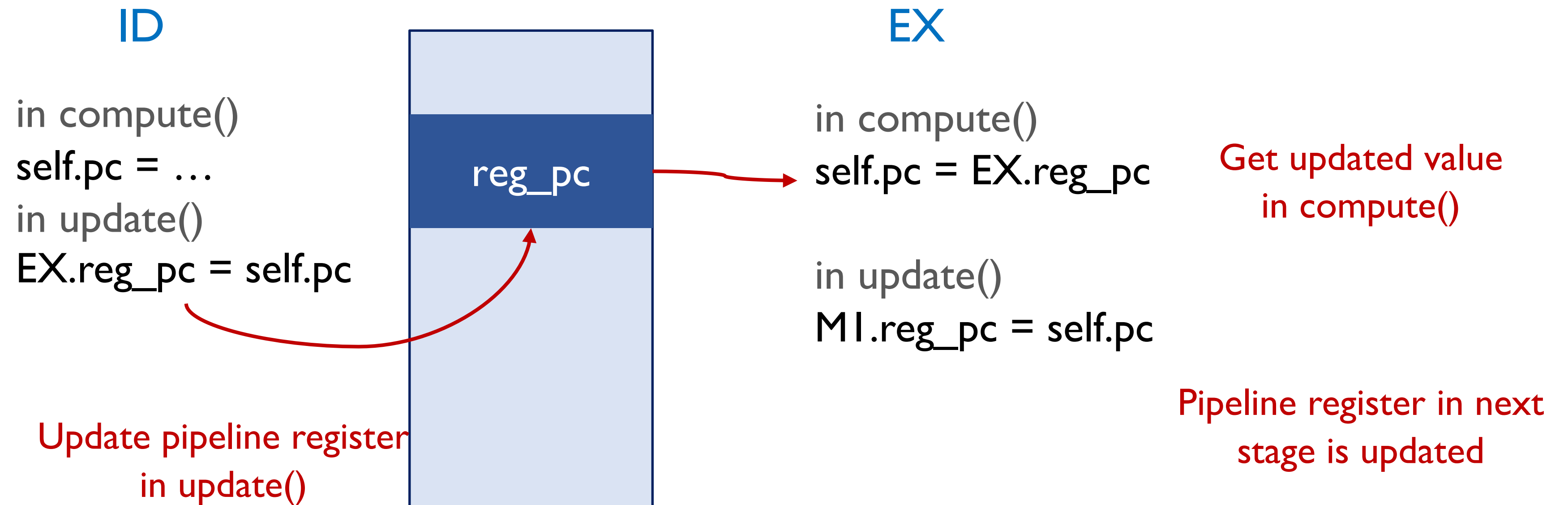
  - Internal signals within a stage
    - Implemented as instance variables
      → referenced as self.[variable name] or Pipe.[class name].[variable name]

    e.g., *self.pc* defined in the ID stage can be referenced as *Pipe.ID.pc*



SNURISC6

- Usage conventions
  - When you want to pass the pipeline register to next stage,



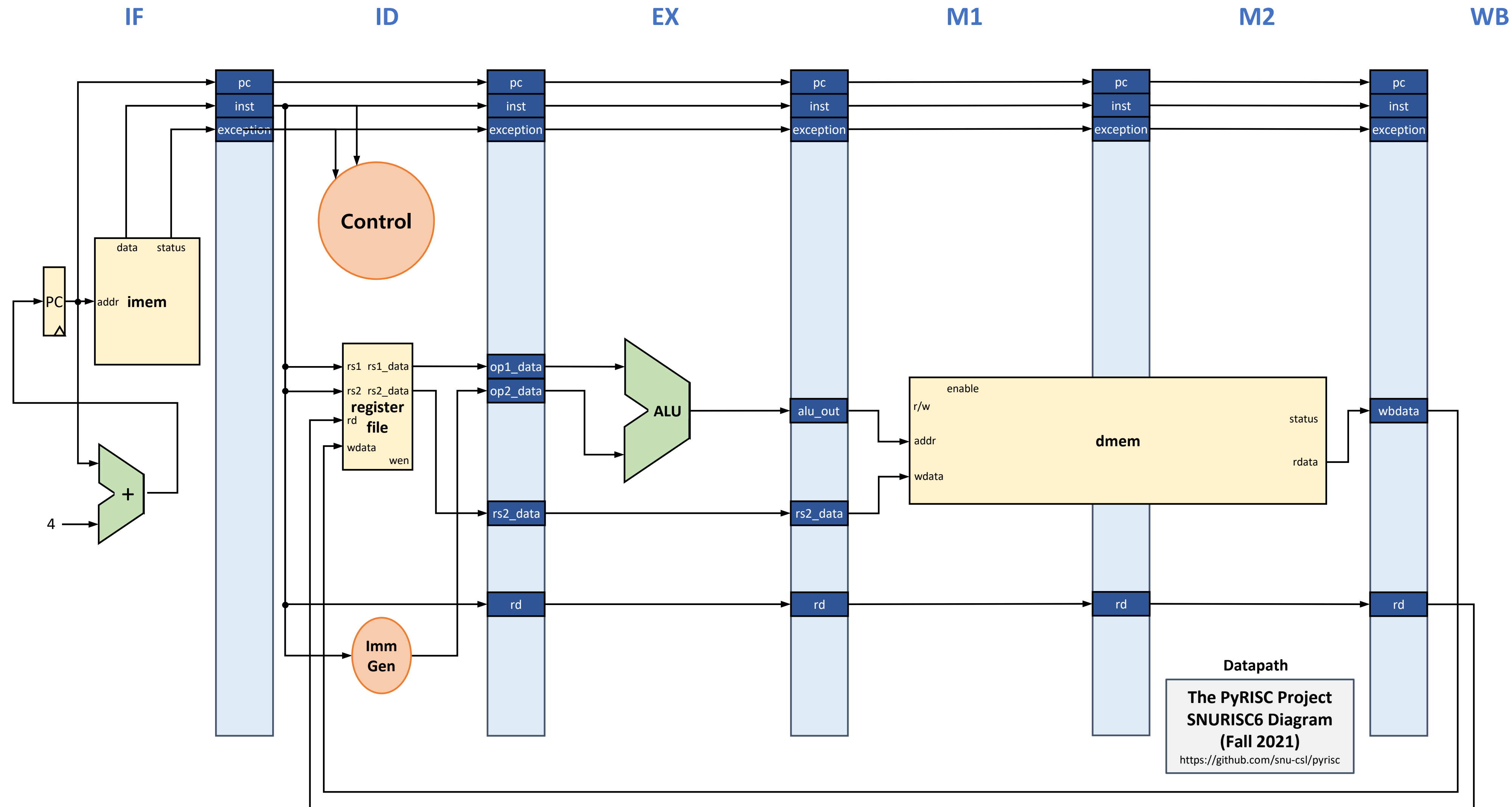
SNURISC6

- For more detailed information, refer to SNURISC5
  - pyrisc/pipe5/README.md
  - pyrisc/pipe5/GUIDE.md

Specification

Skeleton

Currently, it just supports *lw* and *sw* instructions without any hazard detection and control logic



Part 1 (30 Points)

Implementing a 6-stage pipelined RISC-V processor simulator

- It should accept the same RISC-V executable file accepted by SNURISC5
- It should produce same results with SNURISC5
  - In terms of register values and memory states
- Data forwarding should be fully implemented

Part 1 (30 Points)

Implementing a 6-stage pipelined RISC-V processor simulator

- When data forwarding can't solve the dependency among instructions, the pipeline should be stalled
- (M1-M2 hazard) When M1 stage is occupied by *lw* or *sw* instruction, the following *lw* or *sw* instruction should be stalled for one cycle
- You should minimize the number of stalled cycles

Part 1 – Example (1)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lw t0, 0(s0)`	IF	ID	EX	M1	M2	WB				
`addi t0, t0, 1`		IF	ID	ID	ID	EX	M1	M2	WB	
`...`			IF	IF	IF	ID	EX	M1	M2	WB

Data forwarding occurs at the end of M2 stage

Load-use hazard detected → Stall

Load-use hazard
&
Data forwarding is required
&
Data is determined at M2 stage

Part 1 – Example (2)

M1-M2 hazard	1	2	3	4	5	6	7	8	9	10
`lw t0, 0(s0)`	IF	ID	EX	M1	M2	WB				
`lw t1, 4(s0)`		IF	ID	ID	EX	M1	M2	WB		
...			IF	IF	ID	EX	M1	M2	WB	

M1-M2 hazard detected → Stall

Part 2 (30 Points)

Implementing the BTFNT branch prediction scheme

- Backward branch as Taken, Forward branch as Not Taken
  - Instruction in the branch target is immediately fetched for backward branches
  - Instruction next to the branch instruction is fetched for forward branches

- Branch prediction should be performed in the IF stage

- Need to extract the offset value from the instruction word to find out whether it is a forward branch or backward branch

Part 2 (30 Points)

Implementing the BTFNT branch prediction scheme

- Branch outcome is determined in the EX stage

- When the prediction was wrong, you need to cancel the incorrectly fetched instructions and forward the correct value for the next pc

Part 2 (30 Points)

Implementing the BTFNT branch prediction scheme

- You can treat the *jal* instruction as "Always Taken"
- jalr instruction should be handled as "Always-not-Taken" scheme

Part 2 – Example (1)

Forward branch → "Not Taken" branch prediction

Forward, wrong	1	2	3	4	5	6	7	8	9	10
`beq t0, t0, L1`	IF	ID	EX	M1	M2	WB				
`add t1, t2, t3`		IF	ID	BUBBLE	BUBBLE	BUBBLE	BUBBLE			
`addi t1, t1, -1`			IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE	BUBBLE		
`...`										
`...`										
`L1: sub t5, t6, t3`				IF	ID	EX	M1	M2	WB	
`xori t5, t5, 1`					IF	ID	EX	M1	M2	WB
`add t6, t6, t5`						IF	ID	EX	M1	M2

MISPREDICTED

Part 2 – Example (2)

Backward branch → "Taken" branch prediction

Backward, correct	1	2	3	4	5	6	7	8	9	10
`L1: add t1, t2, t3`		IF	ID	EX	M1	M2	WB			
`addi t1, t1, -1`			IF	ID	EX	M1	M2	WB		
`sub t4, t1, t2`				IF	ID	EX	M1	M2	WB	
`...`										
`...`										
`beq t0, t0, L1`	IF	ID	EX	M1	M2	WB				
`sub t5, t6, t3`										
`xori t5, t5, 1`										

Part 2 – Example (3)

“Always-Taken” prediction

jal instruction	1	2	3	4	5	6	7	8	9	10
`jal  ra, L1`	IF	ID	EX	M1	M2	WB				
`add  t1, t2, t3`										
`addi t1, t1, -1`										
`...`										
`...`										
`L1: sub  t5, t6, t3`		IF	ID	EX	M1	M2	WB			
`xori t5, t5,  1`			IF	ID	EX	M1	M2	WB		
`add  t6, t6, t5`				IF	ID	EX	M1	M2	WB	

Part 2 – Example (4)

"Always-not-Taken" prediction

jalr instruction	1	2	3	4	5	6	7	8	9	10
`jalr x0, 0(ra)`	IF	ID	EX	M1	M2	WB				
`add  t1, t2, t3`		IF	ID (MISPREDICTED)	BUBBLE	BUBBLE	BUBBLE	BUBBLE			
`addi t1, t1, -1`			IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE	BUBBLE		
`...`										
(ra contains L1) `...`										
`L1: sub  t5, t6, t3`				IF	ID	EX	M1	M2	WB	
`xori t5, t5, 1`					IF	ID	EX	M1	M2	WB
`add  t6, t6, t5`						IF	ID	EX	M1	M2

Part 3 (20 Points)

Implementing return address stack

- It is difficult to obtain the target address of the *jalr* instruction in the IF stage

- You can solve this problem by using the fact that the *jalr* instruction is mostly used for implementing returns from function calls
  - Target address for the *jalr* instruction was written into the ra register by previous *jal* or *jalr* instruction that invoked the function call

Part 3 (20 Points)

Implementing return address stack

- We introduce the **return address stack** (RAS) to our SNURISC6
  - Small, fixed-size memory of 32-bit return addresses maintained with stack discipline
  - A RAS object with 8-entries is already available when the CPU is initialized

- If RAS provides wrong predicted return address, incorrectly fetched instruction should be handled in the same way as the mispredicted branches

Part 3 (20 Points)

Implementing return address stack

- When rd = x1 for *jal*/*jalr* instruction, current instruction is predicted to make a function call
  - Push the return address (pc + 4) into the RAS

```
Pipe.cpu.ras.push(return_address)
```

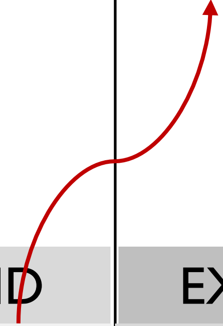
- When rd = x0 & rs1 = x1 & offset = 0 for *jalr* instruction, current instruction is predicted to return from a function call
  - Pop an address from the RAS and use address for the next pc value

```
pc_next, status = Pipe.cpu.ras.pop()
```

Part 3 – Example (1)

RAS provides the correct address

jalr instruction with RAS	1	2	3	4	5	6	7	8	9	10
`L0: jal ra, L1`	IF	ID	EX	M1	M2	WB				
`add t1, t2, t3`	L0+4 pushed to RAS			IF	ID	EX	M1	M2	WB	
`addi t1, t1, -1`					IF	ID	EX	M1	M2	WB
`...`										
`...`										
`L1: sub a0, a0, a1`		IF	ID	EX	M1	M2	WB			
`jalr x0, 0(ra)`			IF	ID	EX	M1	M2	WB		



L0+4 popped from RAS

Part 3 – Example (2)

RAS has address of L0, but ra contains address of L1

jalr instruction with RAS	1	2	3	4	5	6	7	8	9	10
`jalr x0, 0(ra)`	IF	ID	EX	M1	M2	WB				
`add  t1, t2, t3`										
`...`										
`L0: sub  a0, a0, a1`		IF	ID	BUBBLE	BUBBLE	BUBBLE	BUBBLE			
`addi t1, t1, -2`			IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE	BUBBLE		
`...`										
`L1: sub  t5, t6, t3`				IF	ID	EX	M1	M2	WB	
`xori t5, t5,  1`					IF	ID	EX	M1	M2	WB
`add  t6, t6, t5`						IF	ID	EX	M1	M2

MISPREDICTED

Got the actual address L1

L0 popped from RAS

Part 4 (20 Points)

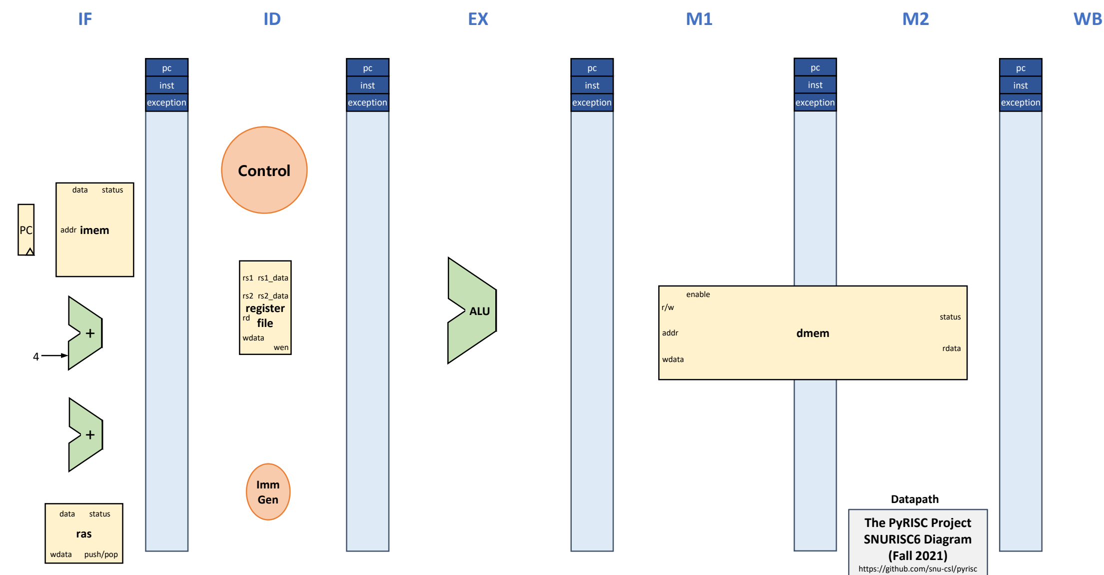
Design document (5 points each)

1. What does the overall pipeline architecture look like?
2. (About Part 1) When do structural/data hazards occur and how do you deal with them?
3. (About Part 2) When do control hazards occur and how do you deal with them with the **BTFNT** branch prediction scheme?
4. (About Part 3) How do you use RAS?

Part 4 (20 Points)

Design document

1. What does the overall pipeline architecture look like?
   - Complete the diagram in snurisc6-design.pdf according to your pipeline design
   - A hand-drawn diagram is OK
   - Take a picture of your diagram and attach it in your design document



Part 4 (20 Points)

Design document

2. When do structural/data hazards occur and how do you deal with them?
   - Specify all the possible conditions when structural/data hazards can occur
   - Show the required control logic to deal with structural/data hazards

Part 4 (20 Points)

Design document

3. When do control hazards occur and how do you deal with them with the **BTFNT** branch prediction scheme?
   - Specify all the possible cases when control hazards can occur
   - Show the required control logic to deal with control hazards

Part 4 (20 Points)

Design document

4. How do you use RAS?
   - Show the required control logic to use the RAS

Specification

- Your task is to modify the stages.py file and make it work correctly for any combination of instructions

- You can test your simulator with RISC-V executable files in *pyrisc/asm/*
  - Highly recommended to write your own test programs to see how your simulator works in a particular situation

```
$ ./snurisc6.py -l 4 [path_to_pyrisc]/pyrisc/asm/sum100
```

Can see various logs depending on the log level

Specification

- You should not change any files other than stages.py
- Your stages.py file should not contain any print() function even in comment lines
- Your simulator should minimize the number of stalled cycles

Specification

- Your code should finish within a reasonable number of cycles
  - If your simulator runs beyond the predefined threshold, you will get TIMEOUT error

- The number of submissions to the server will be limited to 50 times

Submission

- Due: 11:59PM, December 18 (Saturday)
  - 25% of the credit will be deducted for every single day delay
  - This is the final project, so feel free to use all your remaining slip days

- Submit the stages.py file to the submission server
- Also, submit the design document(in PDF file only) to the submission server

Thank You

- Don't forget to read the detailed description before you start your assignment
- If you have any questions about the assignment, feel free to ask via KakaoTalk
- This file will be uploaded after the lab session☺