4190.308: Computer Architecture Lab. 4

Jaehoon Shim
Seongyeop Jeong
Ilkueon Kang
Wookje Han
Jinsol Park

(snucsl.ta@gmail.com)

Systems Software & Architecture Lab.

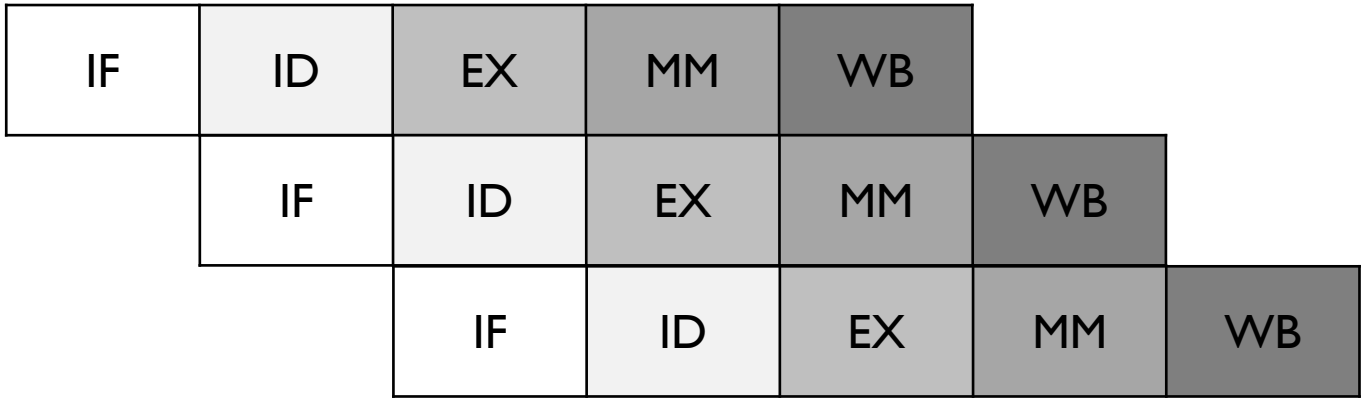
Seoul National University

Fall 2022

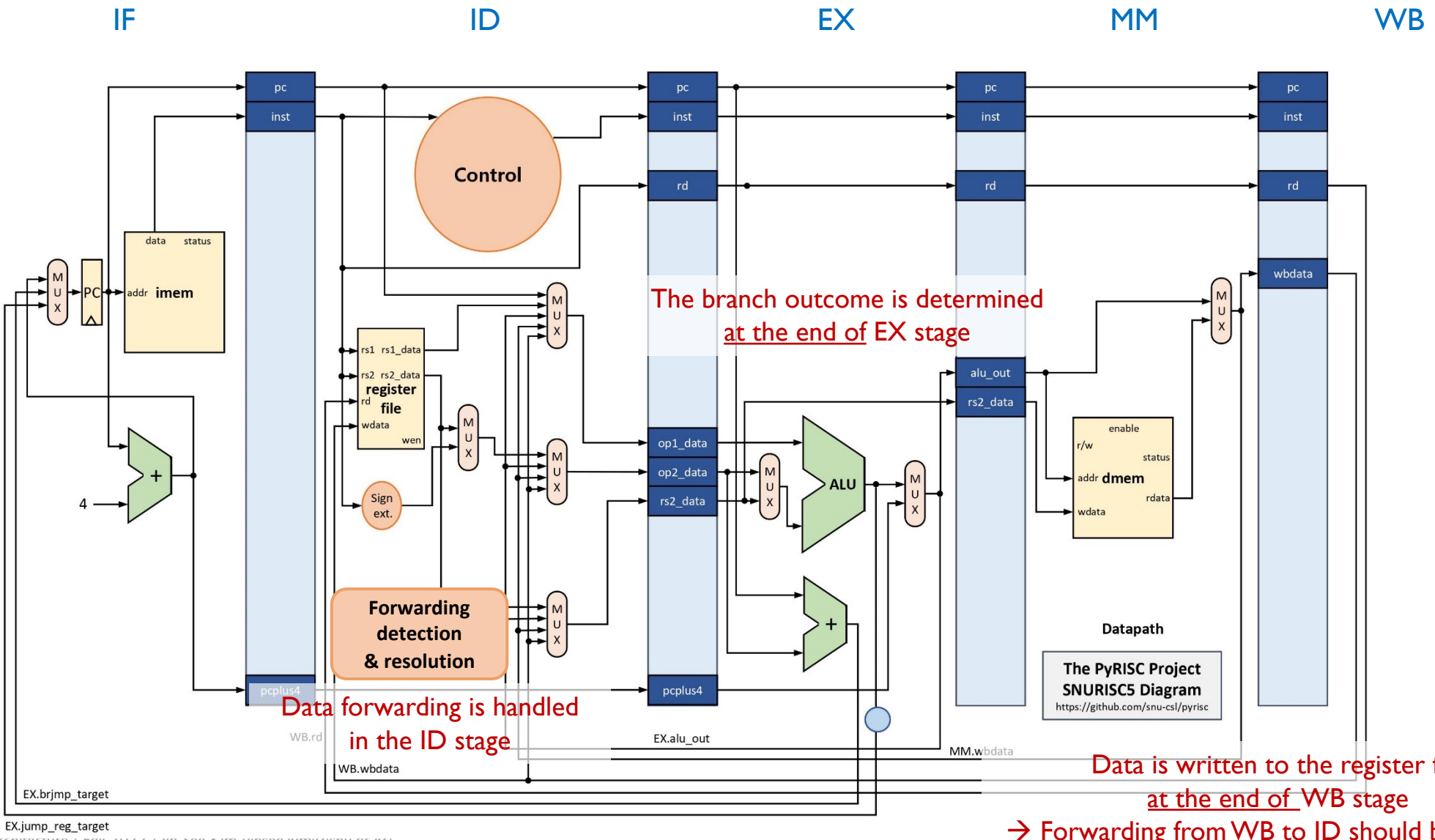
SNURISC5

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



The branch outcome is determined at the end of EX stage

Data forwarding is handled in the ID stage

Data is written to the register file at the end of WB stage

→ Forwarding from WB to ID should be handled

SNURISC5

- Overall simulator architecture
  - snurisc5.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
  - stages.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, and Adder
  - 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

SNURISC5

- class Pipe (in datapath.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.MM = stages[S_MM]
    Pipe.WB = stages[S_WB]
```

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

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

        if not ok:
            break
```

Reverse order due to dependence of hazard/forwarding detection

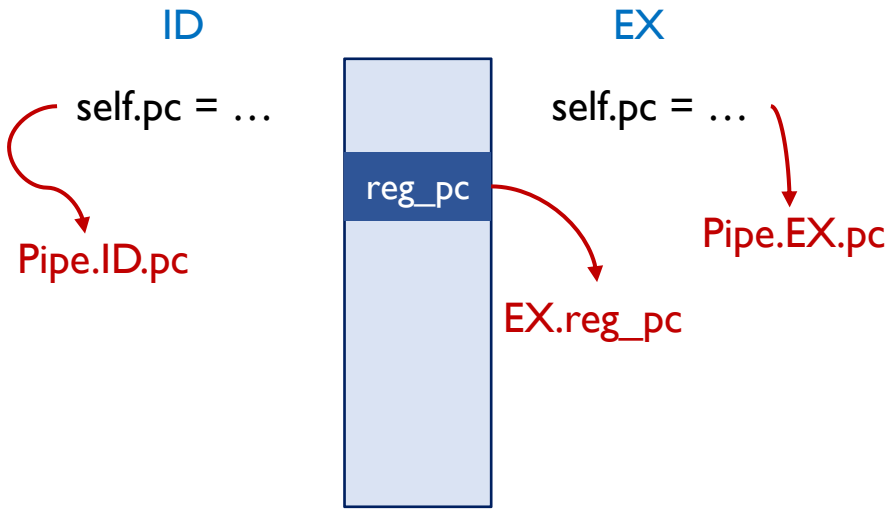
Manipulation of signals using some combinational logic performed inside of the stage

Contents of the pipeline registers are updated

SNURISC5

- 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

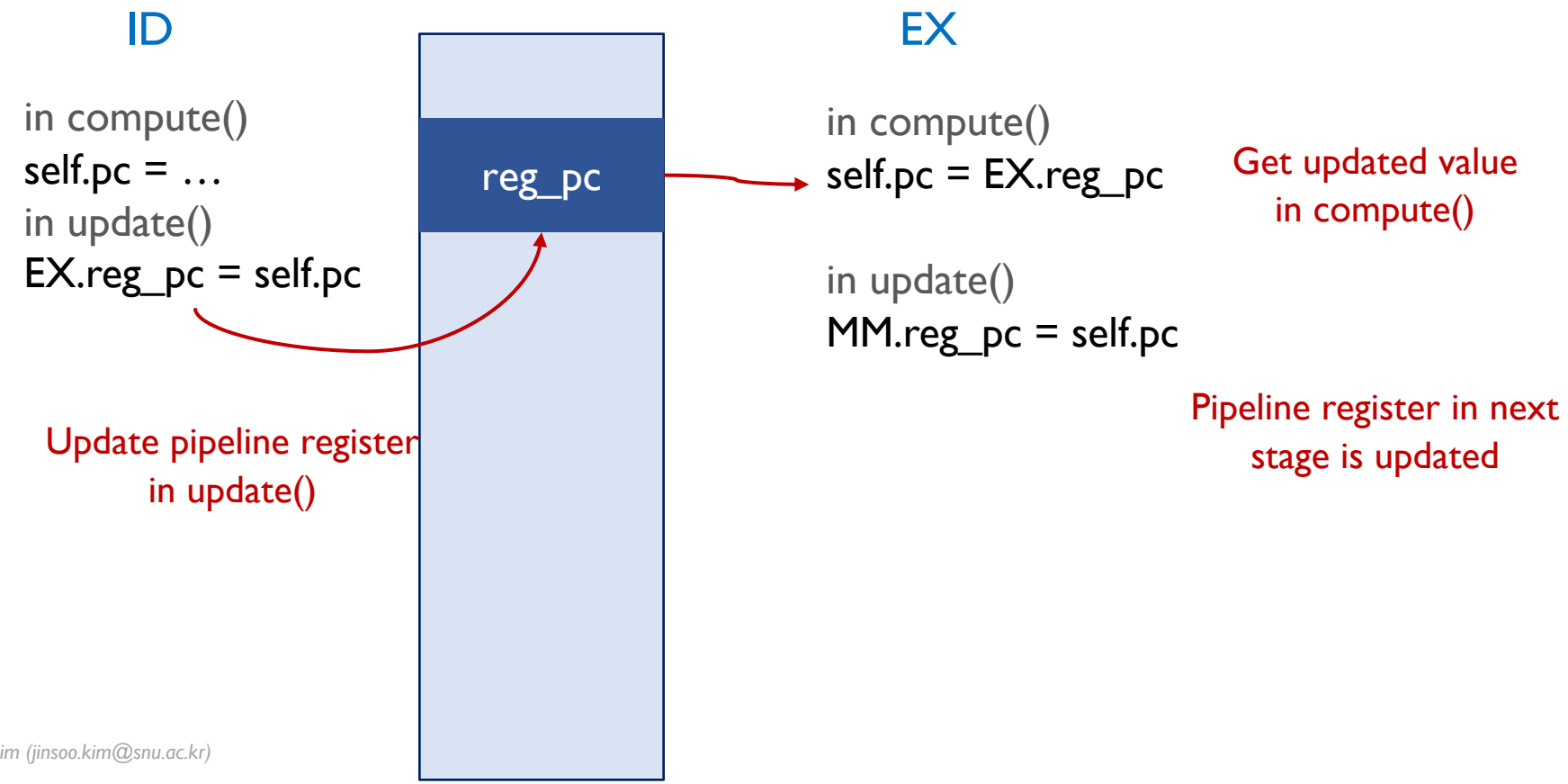


  - Internal signals within a stage
    - Implemented as instance variables
      → referenced as self.[variable name] or Pipe.[class name].[variable name]

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

SNURISC5

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



SNURISC5

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

Specifications

Overview

- [Part 1] Supporting push & pop instructions
- [Part 2] Branch Prediction with Branch Target Buffer (BTB)
- [Part 3] Design Document

[Part 1] Supporting push & pop instructions

Part 1 (40 Points)

Supporting push & pop instructions with 5-stage pipelined RISC-V processor simulator

- Push, Pop: instructions to access data in the stack memory
- Not originally part of the RISC-V architecture set

```
push reg:      R[sp] <- R[sp] – 4
               M[R[sp]] <- R[reg]

pop  reg:      R[reg] <- M[R[sp]]
               R[sp] <- R[sp] + 4
```

Part 1 (40 Points)

- We encoded push & pop instructions in R-type format
  - Argument of push is encoded in the `rs2` field (value will be written to memory like sw instruction)
  - Argument of pop is encoded in the `rd` field

	funct7	rs2	rs1	funct3	rd	opcode
`push x31:`	0 0 0 0 0 0 1	1 1 1 1 1	0 0 0 0 0	0 0 0	0 0 0 0 0	1 1 0 1 0 1 1
`pop x30:`	0 0 0 0 0 1 0	0 0 0 0 0	0 0 0 0 0	0 0 0	1 1 1 1 0	1 1 0 1 0 1 1

Part 1 – Example (1)

Forward sp	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID								
`push t0`		IF								
`pop t1`										

push & pop requires the value of sp before EX stage
→ forwarded

Part 1 – Example (1)

Forward sp	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX							
`push t0`		IF	ID							
`pop t1`			IF							

sp is available now!

for push, sp needs to be decremented at EX stage

push & pop requires the value of sp before EX stage → forwarded

Part 1 – Example (1)

Forward sp	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM						
`push t0`		IF	ID	EX						
`pop t1`			IF	ID						

sp forwarded

decremented sp is available now

push & pop requires the value of sp before EX stage → forwarded

for pop, sp needs to be incremented at EX stage

Part 1 – Example (1)

Forward sp	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push t0`		IF	ID	EX	MM					
`pop t1`			IF	ID	EX					

sp forwarded

sp forwarded

push & pop requires the value of sp before EX stage → forwarded

Part 1 – Example (1)

Forward sp	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push t0`		IF	ID	EX	MM	WB				
`pop t1`			IF	ID	EX	MM	WB			

sp forwarded

sp forwarded

push & pop requires the value of sp before EX stage → forwarded

Part 1 – Example (2)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM						
`push zero`		IF	ID	EX						
`pop t0`			IF	ID						
`push t0`				IF						

sp forwarded

Part 1 – Example (2)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push zero`		IF	ID	EX	MM					
`pop t0`			IF	ID	EX					
`push t0`				IF	ID					

Value is now pushed to stack on dmem

sp forwarded

Part 1 – Example (2)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push zero`		IF	ID	EX	MM	WB				
`pop t0`			IF	ID	EX	MM				
`push t0`				IF	ID	ID				

`t0` can be known only after MM stage of pop is done

Even though `t0` is not needed in the EX stage, we stall just like the load-use hazard

Part 1 – Example (2)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push zero`		IF	ID	EX	MM	WB				
`pop t0`			IF	ID	EX	MM	WB			
`push t0`				IF	ID	ID	EX			

`t0` forwarded

Part 1 – Example (2)

Load-use hazard	1	2	3	4	5	6	7	8	9	10
`lui sp, 0x80020`	IF	ID	EX	MM	WB					
`push zero`		IF	ID	EX	MM	WB				
`pop t0`			IF	ID	EX	MM	WB			
`push t0`				IF	ID	ID	EX	MM	WB	

[Part 2] Branch Prediction with Branch Target Buffer

Part 2 (40 Points)

Branch Prediction with Branch Target Buffer (BTB)

- Current pipelined processor uses an ***always-not-taken*** branch prediction scheme
- Two cycles are wasted when the branch is taken

Always-not-taken	1	2	3	4	5	6	7	8	9	10
`beq t0, t0, L1`	IF	ID	EX	MM	WB					
`add t1, t2, t3`		IF	ID	BUBBLE	BUBBLE	BUBBLE				
`addi t1, t1, -1`			IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE			
`...`										
`...`										
`L1: sub t5, t6, t3`				IF	ID	EX	MM	WB		
`xori t5, t5, 1`					IF	ID	EX	MM	WB	
`add t6, t6, t5`						IF	ID	EX	MM	WB

MISPREDICTED

Branch is always predicted to be not-taken, so just fetch instructions from PC+4

Part 2 (40 Points)

Branch Prediction with Branch Target Buffer (BTB)

- Use a **branch predictor** to increase the chances of fetching the right instruction after a branch

Right Prediction	1	2	3	4	5	6	7	8	9	10
`beq t0, t0, L1`	IF	ID	EX	MM	WB					
`add t1, t2, t3`	Predict that branch will be taken in the IF stage!									
`addi t1, t1, -1`										
`...`	Fetch the right instruction from the target address									
`...`										
`L1: sub t5, t6, t3`		IF	ID	EX	MM	WB				
`xori t5, t5, 1`			IF	ID	EX	MM	WB			
`add t6, t6, t5`				IF	ID	EX	MM	WB	No cycles wasted! :)	

Part 2 (40 Points)

The Branch Target Buffer (BTB)

- Caches recent information on **taken** branches
  - Valid bit: 0 or 1. Indicates whether the corresponding entry has valid information
  - Tag: Represent the address of a branch instruction
  - Target Address: Address to jump to when the branch is taken.

N = 2^k entries

	Valid Bit	Tag	Target Address
Index 0	…	…	…
Index N-1			

Part 2 (40 Points)

1) Encounter a branch instruction (IF stage)

```
PC → 0x80000004        beq  t0, t0, L1
     0x80000008        add  t1, t2, t3
     0x8000000c        addi t1, t1, -1
                  ...
     0x80000024    L1: sub  t5, t6, t3
```

Part 2 (40 Points)

Last two bits not considered because they are always 0b00

1) Encounter a branch instruction (IF stage)

```
PC → 0x80000004        beq  t0, t0, L1
     0x80000008        add  t1, t2, t3
     0x8000000c        addi t1, t1, -1
                  ...
     0x80000024   L1:  sub  t5, t6, t3
```

2) Calculate the tag and index from the PC

Remaining 32-(k+2) bits → this becomes the **tag**

0b1000 0000 0000 0000 0000 0000 0000 01~~00~~

If k=4 → this becomes the **index**

Part 2 (40 Points)

Last two bits not considered because they are always 0b00

1) Encounter a branch instruction (IF stage)

```
PC → 0x80000004        beq  t0, t0, L1
     0x80000008        add  t1, t2, t3
     0x8000000c        addi t1, t1, -1
                  ...
     0x80000024   L1:  sub  t5, t6, t3
```

2) Calculate the tag and index from the PC

Remaining 32-(k+2) bits → this becomes the **tag**

0b1000 0000 0000 0000 0000 0000 0000 01~~00~~

If k=4 → this becomes the **index**

3) Check the corresponding entry in the BTB

$N = 2^k$

	Valid Bit	Tag	Target Address
Index 0			
Index 1	1	0x2000000	0x80000024
…	…	…	…
Index N-1			

Part 2 (40 Points)

Last two bits not considered because they are always 0b00

1) Encounter a branch instruction (IF stage)

```
PC → 0x80000004        beq  t0, t0, L1
     0x80000008        add  t1, t2, t3
     0x8000000c        addi t1, t1, -1
                  ...
     0x80000024   L1:  sub  t5, t6, t3
```

2) Calculate the tag and index from the PC

Remaining 32-(k+2) bits → this becomes the **tag**

0b1000 0000 0000 0000 0000 0000 0000 0100

If k=4 → this becomes the **index**

3) Check the corresponding entry in the BTB

N = 2^k

	Valid Bit	Tag	Target Address
Index 0			
Index 1	1	0x2000000	0x80000024
…	…	…	…
Index N-1			

4) Fetch next instruction from target address

```
     0x80000004        beq  t0, t0, L1
     0x80000008        add  t1, t2, t3
     0x8000000c        addi t1, t1, -1
                  ...
PC → 0x80000024   L1:  sub  t5, t6, t3
```

Part 2 – Example (1)

IF	ID	EX	MM	WB
bne	addi	li		

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`0x80000000 li t0, 3`	IF	ID	EX												
`0x80000004 L0: addi t0, t0, -1`		IF	ID												
`0x80000008 bne t0, x0, L0`			IF												
`0x8000000c sub t1, t0, x0`															
`0x8000000f ebreak`															

BTB is empty, so just fetch instructions from PC+4

Index	Valid	Tag	Target Addr

K=4

IF	ID	EX	MM	WB
sub	bne	addi	li	

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM											
0x80000004	L0: addi t0, t0, -1		IF	ID	EX											
0x80000008	bne t0, x0, L0			IF	ID											
0x8000000c	sub t1, t0, x0				IF											
0x8000000f	ebreak															

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
ebrk	sub	bne	addi	li

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
`0x80000000 li t0, 3`	IF	ID	EX	MM	WB										
`0x80000004 L0: addi t0, t0, -1`		IF	ID	EX	MM										
`0x80000008 bne t0, x0, L0`			IF	ID	EX										
`0x8000000c sub t1, t0, x0`				IF	ID										
`0x8000000f ebreak`					IF										

At EX stage of bne, we know that the branch is taken

1) Update the BTB entry
2) Flush instructions in the IF and ID stages

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
addi	BUB	BUB	bne	addi

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF									
0x80000008	bne t0, x0, L0			IF	ID	EX	MM									
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE									
0x8000000f	ebreak					IF	BUBBLE									

Next instruction is fetched from the target address

Incorrectly fetched instructions are changed into BUBBLEs

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
bne	addi	BUB	BUB	bne

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID								
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF								
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE								
0x8000000f	ebreak					IF	BUBBLE	BUBBLE								

At IF stage - Looking into the BTB with PC 0x80000008…

1) Index is 2 → get the BTB entry from that index
2) Valid bit is 1 !!
3) Tag matches !!

Okay let's fetch instruction from 0x80000004 !!

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
addi	bne	addi	BUB	BUB

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX							
									IF	Instruction is fetched from 0x80000004 according to the prediction						
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID							
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE							

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
bne	addi	bne	addi	BUB

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX	MM						
									IF	ID						
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID	EX						
										IF						
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE						

bne is in the EX stage. We are able to determine that…

1) The branch is actually taken (t0 != x0)
2) The target address (L0) is the same as what we predicted using the BTB

Okay we do not flush instructions in the IF and ID stages!

Meanwhile, we look at the BTB, and fetch the next instruction from 0x80000004.

Index	Valid	Tag	Target Addr
2	1	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
addi	bne	addi	bne	addi

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX	MM	WB					
									IF	ID	EX					
											IF					
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID	EX	MM					
										IF	ID					
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE						

Instruction is fetched from 0x80000004 according to the prediction

Index	Valid	Tag	Target Addr
2	0	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
bne	addi	bne	addi	bne

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX	MM	WB					
									IF	ID	EX	MM				
											IF	ID				
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID	EX	MM	WB				
										IF	ID	EX				
												IF				
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE						

At this point, we find out that our prediction was wrong (i.e. t0 == x0)
We change the valid bit in the BTB to 0, and flush instructions in the IF and ID stages.

Index	Valid	Tag	Target Addr
2	0	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
sub	BUB	BUB	bne	addi

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX	MM	WB					
									IF	ID	EX	MM	WB			
											IF	ID	BUBBLE			
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID	EX	MM	WB				
										IF	ID	EX	MM			
												IF	BUBBLE			
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
													IF	Fetch the next instruction properly		
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE						

Index	Valid	Tag	Target Addr
2	0	0x2000000	0x80000004

K=4

IF	ID	EX	MM	WB
-	-	ebrk	sub	BUB

		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0x80000000	li t0, 3	IF	ID	EX	MM	WB										
0x80000004	L0: addi t0, t0, -1		IF	ID	EX	MM	WB									
							IF	ID	EX	MM	WB					
									IF	ID	EX	MM	WB			
											IF	ID	BUBBLE	BUBBLE	BUBBLE	
0x80000008	bne t0, x0, L0			IF	ID	EX	MM	WB								
								IF	ID	EX	MM	WB				
										IF	ID	EX	MM	WB		
												IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE
0x8000000c	sub t1, t0, x0				IF	ID	BUBBLE	BUBBLE	BUBBLE							
													IF	ID	EX	MM
0x8000000f	ebreak					IF	BUBBLE	BUBBLE	BUBBLE	BUBBLE						

Restrictions

- The BTB is checked in the IF stage in the `compute(self)` function
- The BTB is updated in the EX stage in the `update(self)` function

- There can be a case when the fetched instruction is correct, even though the prediction was wrong (e.g. jumping to PC+4)
  - In such cases, even though it is inefficient, flush the instructions in the IF and ID stages.

[Part 3] Design Document

Part 3 (20 Points)

Design Document for your modifications

- About Part 1: When do the new data hazards occur due to the push and pop instructions and how do you deal with them?
- About Part 2: How do you implement the branch prediction using the BTB?

- Refer to the README for further specifications.

How build GNU toolchain

PyRISC

- It provides various RISC-V toolset written in Python
- It has snurisc, a RISC-V instruction set simulator that supports most of RV32I base instruction set (32-bit version!)
- You should work on either Linux or MacOS
  - We highly recommend you to use Ubuntu 18.04 LTS or later
- For Windows, we recommend installing WSL(Windows Subsystem for Linux) and Ubuntu

PyRISC Prerequisites

- PyRISC toolset requires Python version 3.6 or higher.
- You should install Python modules(numpy, pyelftools)

For Ubuntu 18.04 LTS,

```
$ sudo apt-get install python3-numpy python3-pyelftools
```

For MacOS,

```
$ pip install numpy pyelftools
```

RISC-V GNU toolchain

- In order to work with the PyRISC toolset, you need to build a RISC-V GNU toolchain for the RV32I instruction set
- We have added a patch to enable push and pop instructions

- Please take the following steps to build it on your machine

Building RISC-V GNU toolchain

1. Install prerequisite packages

For Ubuntu 18.04 LTS,

```
$ sudo apt-get install autoconf automake autotools-dev curl libmpc-dev
$ sudo apt-get install libmpfr-dev libgmp-dev gawk build-essential bison flex
$ sudo apt-get install texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev
```

For MacOS,

```
$ brew install gawk gnu-sed gmp mpfr libmpc isl zlib expat
```

Building RISC-V GNU toolchain

2. Download the RISC-V GNU Toolchain from Github

```
$ git clone --recursive https://github.com/riscv/riscv-gnu-toolchain
```

3. Use the patch to enable push & pop instructions

```
$ cp ~/ca-pa4/patch/pushpop.patch ~/riscv-gnu-toolchain
$ cd ~/riscv-gnu-toolchain
$ patch -p1 < ./pushpop.patch
```

Building RISC-V GNU toolchain

4. Configure the RISC-V GNU toolchain

```
$ cd riscv-gnu-toolchain
$ mkdir build
$ cd build
$ ../configure --prefix=/opt/riscv --with-arch=rv32i
```

5. Compile and install them

```
$ sudo make
```

6. Add `/opt/riscv/bin` in your PATH

```
$ export PATH=/opt/riscv/bin:$PATH
```

Skeleton & How to run

Skeleton

- We provide you with push and pop encodings and masks in `isa.py`

```
# Instruction Encodings
PUSH = WORD(0b00000010000000000000000001101011)
POP  = WORD(0b00000100000000000000000001101011)

# Instruction Masks
PUSH_MASK = WORD(0b11111110000000000111000001111111)
POP_MASK  = WORD(0b11111110000000000111000001111111)

# ISA table
PUSH : [ "push", PUSH_MASK, R_TYPE, CL_MEM, ]
POP  : [ "pop", POP_MASK, R_TYPE, CL_MEM, ]
```

Running RISC-V executable file

- To run test codes
  - Make the assembly test cases in `asm/` into an executable file
  - Use the –l option for log level, and –b option for defining k for BTB

```
$ cd asm/ && make && cd ../
$ ./snurisc5.py -l 4 -b 7 asm/ex4
```

Restrictions

- You should not change any files other than stages.py
- Your stages.py file should <u>not contain any print() function even in comment lines</u>
- You should not introduce unnecessary pipeline stalls.
- Your code should finish within a reasonable number of of cycles.
  - If your simulator runs beyond the predefined threshold, you will get the TIMEOUT error.

Submission

- Due: 11:59PM, December 18 (Sunday)
  - The **sys server will be closed at 11:59PM on December 22nd**. This is the firm deadline.
  - This is the final project, so feel free to use all your remaining slip days
  - Only the upload submitted before the deadline will receive the full credit. 25% of the credit will be deducted for every single day delay.

- Submit the stages.py file to the submission server
- Also, submit the design document(in PDF file only) to the submission server
- The submitted code will NOT be graded instantly. Instead, it will be graded every four hours (12:00am, 4:00am, 8:00am, 12:00pm, 4:00pm, 8:00pm). You may submit multiple versions, but only the last version will be graded.

Logistics

- Part 1 (40) + Part 2 (40) + Part 3 (20) = 100
- Overall project logistics of this course:
  - Proj 1 (3%) + Proj 2 (8%) + Proj 3 (14%) + Proj 4 (15%) = 40% of the entire course

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☺