CS 61C Summer 2022 — RISC-V Single Cycle Datapath — Discussion 7

1 Pre-Check

This section is designed as a conceptual check for you to determine if you conceptually understand and have any misconceptions about this topic. Please answer true/false to the following questions, and include an explanation:

1.1 The single cycle datapath makes use of all hardware units for each instruction.

False. All units are active in each cycle, but their output may be ignored (gated) by control signals.

1.2 It is possible to execute the stages of the single cycle datapath in parallel to speed up execution of a single instruction.

False. Each stage depends on the value produced by the stage before it (e.g., instruction decode depends on the instruction fetched).

1.3 The auipc instruction and jump instructions (jal, jalr, and any pseudoinstruction) are the only instructions that set PC = PC + offset.

False. Branch instructions also set PC = PC + offset if a branch condition is met.

1.4 Storing instructions and loading instructions are the only instructions that actively require going to and from DMEM.

True. For all other instructions, we don't need to read the data that is read out from DEMEM, and thus don't need to wait for the output of the MEM stage.

1.5 It is possible to use both the output of the immediate generator and the value in register rs2.

False. You may only use *either* the immediate generator or the value in register rs2. Notice in our datapath, there is a mux with a signal (BSel) that decides whether we use the output of the immediate generator or the value in rs2.

1.6 Combinational logic is only used in the instruction decode stage.

False. Other stages executes combinational logic too (muxes for instruction fetch, memory write, register updates; ALU operations during execute).

2 Single-Cycle CPU

2.1 For this worksheet, we will be working with the single-cycle CPU datapath on the last page.

(a) Explain what happens in each datapath stage, and which hardware units in the datapath are used.

IF Instruction Fetch

Send address to the instruction memory (IMEM), and read IMEM at that address.
Hardware units: PC register, +4 adder, PCSel mux, IMEM

ID Instruction Decode

Generate control signals from the instruction bits, generate the immediate, and read registers from the RegFile.
Hardware units: RegFile, ImmGen

EX Execute

Perform ALU operations, and do branch comparison.
Hardware units: ASel mux, BSel mux, branch comparator, ALU

MEM Memory

Read from or write to the data memory (DMEM).
Hardware units: DMEM

WB Writeback

Write back either PC + 4, the result of the ALU operation, or data from memory to the RegFile.
Hardware units: WBSel mux, RegFile

(b) On the datapath, fill in each **round** box with the name of the datapath component, and each **square** box with the name of the control signal.

(c) List all possible signals that each control signal may take on for the single cycle datapath. Then mark which ones are actively used. If there are any non-used signals, write a short explanation for why it exists but is not used.

Signal Name	Values	Signal Name	Values
PCSel		RegWEn	
ImmSel		BrEq	
BrLt		ALUSel	
MemRW		WBSel	

PCSel: 0: OldPC + 4 is next PC; 1: ALU value (for branches, jumps, etc...)
RegWEn: 0: WB value cannot be written to regsiter; 1: allowed write
ImmSel: 0-5 used for I, B, S, J, U, and I* type immediates; 6-7 unused
BrEq: 0 when inputs not equal; 1 when inputs equal
BrLt: 0 when `rs1` is not less than `rs2`; 1 when it is less than
ALUSel: note, this is using the same design reference 61C does; may differ based on CPU design; you do **not** have to memorise all of these!

- 0: add (`rd = rs1 + rs2`)
- 1: sll (`rd = rs1 << rs2`)
- 2: slt (`rd = (rs1 < rs1 (signed)) ? 1 : 1`)
- 3: unused
- 4: xor (`rd = rs1` $^{r}s2)5 : srl($`rd = (unsigned) A >> B`)
- 6: or (`rd = rs1 | rs2`)
- 7: and (`rd = rs1  rs2`)
- 8: mul (`rd = (signed) (rs1 * rs2)[31:0]`)
- 9: mulh (`rd = (signed) (rs1 * rs2) [63:32]`)
- 10: unused
- 11: mulhu (`rd = (rs1 * rs2) [63:32]`)
- 12: sub (`rd = rs1 - rs2`
- 13: sra (`rd = (signed) rs1 >> rs2`)
- 14: unused
- 15: bsel (`rd = rs2`)

MemRW: 0 for all non-write-to-memory operations; 1 to enable writing to main memory
WBSel: 0 for PC + 4; 1 for ALU output; 2 for main memory read output; 3 unused

2.2 Fill out the following table with the control signals for each instruction based on the datapath on the previous page. If the value of the signal does not affect the execution of an instruction, i.e. the correct execution will still occur, you may use the * (don't care) symbol or write **all** the possible values (e.g. 0/1 for standard datapath signals, or 0/1/2/3 for the WBSel).

Original phrasing: Wherever possible, use * to indicate that what this signal is does not matter (as in, letting the value be whatever it wants won't affect the execution of the instruction). If the value of the signal does matter for correct execution, but can vary, list all of the values (for example, for a signal that matters with possible values of 0 and 1, write 0/1).

	BrEq	BrLT	PCSel	ImmSel	BrUn	ASel	BSel	ALUSel	MemRW	RegWEn	WBSel
add	*	*	0 (PC + 4)	*	*	0 (Reg)	0 (Reg)	add	0	1	1 (ALU)
ori	*	*	0	I	*	0 (Reg)	1 (Imm)	or	0	1	1 (ALU)
lw	*	*	0	I	*	0 (Reg)	1 (Imm)	add	0	1	2 (MEM)
sw	*	*	0	S	*	0 (Reg)	1 (Imm)	add	1	0	*
beq	1/0	*	1/0	SB	*	1 (PC)	1 (Imm)	add	0	0	*
jal	*	*	1 (ALU)	UJ	*	1 (PC)	1 (Imm)	add	0	1	0 (PC + 4)
bltu	*	1/0	1/0	SB	1	1 (PC)	1 (Imm)	add	0	0	*

2.3 Clocking Methodology

- A **state element** is an element connected to the clock (denoted by a triangle at the bottom). The **input signal** to each state element must stabilize before each **rising edge**.
- The **critical path** is the longest delay path between state elements in the circuit. The circuit cannot be clocked faster than this, since anything faster would mean that the correct value is not guaranteed to reach the state element in the alloted time. If we place registers in the critical path, we can shorten the period by **reducing the amount of logic between registers**.

For this exercise, the delay for each circuit element is given as follows:

Clk-to-Q	RegFile Read	RegFile Setup	Mux
5ns	35ns	20ns	15ns

ALU	Branch Comp	Imm Gen	MEM Read	MEM Write
100ns	50ns	45ns	300ns	200ns

(a) Mark the stages of the datapath that the following instructions use

	IF	ID	EX	MEM	WB	
add	X	X	X		X	
ori	X	X	X		X	
lw	X	X	X	X	X	
sw	X	X	X	X		
beq	X	X	X			
jal	X	X	X		X	

(b) Assume the RegFile setup and PC setup times are equivalent. Ignoring the length of a clock cycle, how long does it take to execute the instruction:

1. jal

jal = clk-to-Q + Mem-Read + Imm-Gen + Mux + ALU + max(Mux(WBSel) + RegFileSetup, Mux(PCSel) + PCSetup)
= 5ns + 300ns + 45ns + 15ns + 100ns + 35ns = 500ns

2. lw

lw = clk-to-Q + Mem-Read + max(Mux + RegFileRead, Mux + Imm-Gen) + ALU + Mem-Read + Mux + RegFileSetup
= 5ns + 300ns + 60ns + 100ns + 300ns + 15ns + 20ns = 800ns

3. sw

sw = clk-to-Q + Mem-Read + max(Mux + RegFileRead, Mux + Imm-Gen) + ALU + Mem-Write
= 5ns + 300 ns + 60ns + 100ns + 200ns = 665ns

(c) Which instruction(s) exercise the critical path?

Load word (lw), which uses all 5 stages and takes 800ns.

(d) What is the fastest you could clock this single cycle datapath?

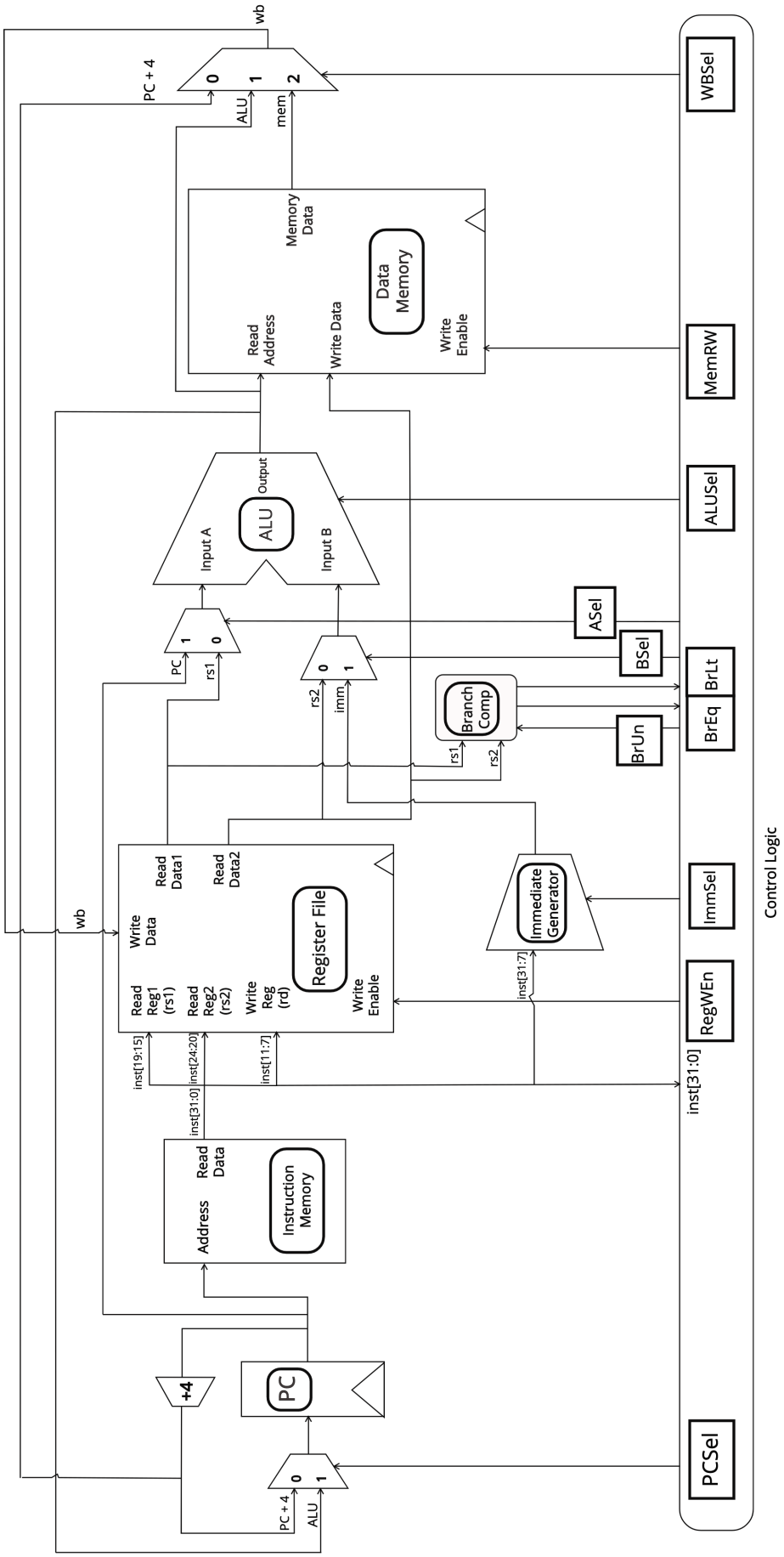
$$\frac{1}{800} \text{ nanoseconds} = \frac{1}{800 * 10^{-9}} \text{ seconds} = 1,250,000 s^{-1} = 0.00125 GHz$$

(e) Why is the single cycle datapath inefficient?

At any given time, most of the parts of the single cycle datapath are sitting unused. Also, even though not every instruction exercises the critical path, the datapath can only be clocked as fast as the slowest instruction.

(f) How can you improve its performance? What is the purpose of pipelining?

Performance can be improved with pipelining, or putting registers between stages so that the amount of combinational logic between registers is reduced, allowing for a faster clock time.


wb
PC + 4
ALU
mem
0
1
2
Memory Data
Data Memory
Read Address
Write Data
Write Enable
Input A
Input B
Output
ALU
PC
rs1
rs2
imm
Branch Comp
rs1
rs2
Read Data1
Read Data2
Write Data
Register File
Read Reg1 (rs1)
Read Reg2 (rs2)
Write Reg (rd)
Write Enable
Immediate Generator
inst[31:7]
inst[19:15]
inst[24:20]
inst[11:7]
inst[31:0]
Address
Read Data
Instruction Memory
+4
PC
PC + 4
ALU
WBSel
MemRW
ALUSel
ASel
BSel
BrLt
BrEq
BrUn
ImmSel
RegWEn
PCSel
inst[31:0]
Control Logic