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

2 Single-Cycle CPU

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.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Values</th>
<th>Signal Name</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCSel</td>
<td></td>
<td>RegWEn</td>
<td></td>
</tr>
<tr>
<td>ImmSel</td>
<td></td>
<td>BrEq</td>
<td></td>
</tr>
<tr>
<td>BrLt</td>
<td></td>
<td>ALUSel</td>
<td></td>
</tr>
<tr>
<td>MemRW</td>
<td></td>
<td>WBSel</td>
<td></td>
</tr>
</tbody>
</table>
**PCSel:** 0: OldPC + 4 is next PC; 1: ALU value (for branches, jumps, etc...)

**RegWEn:** 0: WB value cannot be written to register; 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 \( rd = rs1 \ll rs2 \)
- 2: slt \( rd = (rs1 < rs1 \text{ (signed)}) ? 1 : 1 \)
- 3: unused
- 4: xor \( rd = rs1 ^{rs2} \)
- 5: srl \( rd = (\text{unsigned}) A \gg B \)
- 6: or \( rd = rs1 | rs2 \)
- 7: and \( rd = rs1 \& rs2 \)
- 8: mul \( rd = (\text{signed}) (rs1 \ast rs2)[31:0] \)
- 9: mulh \( rd = (\text{signed}) (rs1 \ast rs2)[63:32] \)
- 10: unused
- 11: mulhu \( rd = (rs1 \ast rs2)[63:32] \)
- 12: sub \( rd = rs1 - rs2 \)
- 13: sra \( rd = (\text{signed}) rs1 \gg rs2 \)
- 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).
<table>
<thead>
<tr>
<th></th>
<th>BrEq</th>
<th>BrLT</th>
<th>PCSel</th>
<th>ImmSel</th>
<th>BrUn</th>
<th>ASel</th>
<th>BSel</th>
<th>ALUSel</th>
<th>MemRW</th>
<th>RegWEn</th>
<th>WB Sel</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>*</td>
<td>*</td>
<td>0 (PC + 4)</td>
<td>*</td>
<td>*</td>
<td>0 (Reg)</td>
<td>0 (Reg)</td>
<td>add</td>
<td>0</td>
<td>1</td>
<td>1 (ALU)</td>
</tr>
<tr>
<td>ori</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>I</td>
<td>*</td>
<td>0 (Reg)</td>
<td>1 (Imm)</td>
<td>or</td>
<td>0</td>
<td>1</td>
<td>1 (ALU)</td>
</tr>
<tr>
<td>lw</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>I</td>
<td>*</td>
<td>0 (Reg)</td>
<td>1 (Imm)</td>
<td>add</td>
<td>0</td>
<td>1</td>
<td>2 (MEM)</td>
</tr>
<tr>
<td>sw</td>
<td>*</td>
<td>*</td>
<td>0</td>
<td>S</td>
<td>*</td>
<td>0 (Reg)</td>
<td>1 (Imm)</td>
<td>add</td>
<td>1</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>beq</td>
<td>1/0</td>
<td>*</td>
<td>1/0</td>
<td>SB</td>
<td>*</td>
<td>1 (PC)</td>
<td>1 (Imm)</td>
<td>add</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
<tr>
<td>jal</td>
<td>*</td>
<td>*</td>
<td>1 (ALU)</td>
<td>UJ</td>
<td>*</td>
<td>1 (PC)</td>
<td>1 (Imm)</td>
<td>add</td>
<td>0</td>
<td>1</td>
<td>0 (PC + 4)</td>
</tr>
<tr>
<td>bltu</td>
<td>*</td>
<td>1/0</td>
<td>1/0</td>
<td>SB</td>
<td>1</td>
<td>1 (PC)</td>
<td>1 (Imm)</td>
<td>add</td>
<td>0</td>
<td>0</td>
<td>*</td>
</tr>
</tbody>
</table>
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 allotted 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:

<table>
<thead>
<tr>
<th>Element</th>
<th>Delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>Clk-to-Q</td>
<td>5 ns</td>
</tr>
<tr>
<td>RegFile Read</td>
<td>35 ns</td>
</tr>
<tr>
<td>RegFile Setup</td>
<td>20 ns</td>
</tr>
<tr>
<td>Mux</td>
<td>15 ns</td>
</tr>
<tr>
<td>ALU</td>
<td>100ns</td>
</tr>
<tr>
<td>Branch Comp</td>
<td>50 ns</td>
</tr>
<tr>
<td>Imm Gen</td>
<td>45 ns</td>
</tr>
<tr>
<td>MEM Read</td>
<td>300 ns</td>
</tr>
<tr>
<td>MEM Write</td>
<td>200 ns</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>ori</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>lw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>jal</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td></td>
<td>X</td>
</tr>
</tbody>
</table>

(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)
= 5 ns + 300 ns + 45 ns + 15 ns + 100 ns + 35 ns = 500 ns

2. lw

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

3. sw

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

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

(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.