# CS151B Spring 2017 Homework #4 Solutions

**Problem (1)**

**A)**

$$T_{exec} = IC \times CPI \times T_{cycle}$$

Hence,

$$CPI = \frac{T_{exec}}{IC \times T_{cycle}}$$

CPI for code compiled by A =

$$\frac{1.1}{10^9 \times 10^{-9}} = 1.1$$

CPI for code compiled by B =

$$\frac{1.5}{1.2 \times 10^9 \times 10^{-9}} = 1.25$$

**B)** The execution time of processor running the compiled code is:

$$T_{exec} = CPI \times IC \times T_{cycle}$$

As the execution times for the two processors are the same, we have:

$$CPI_A \times IC_A \times T_{cycleA} = CPI_B \times IC_B \times T_{cycleB}$$

When reporting how much "faster" is some X compared to Y, we are dealing with a throughput measure, for which higher is "better". In this case, the measure of interest is the clock frequency.

$$R = \frac{F_A}{F_B} = \frac{T_{cycleB}}{T_{cycleA}} = \frac{CPI_A \times IC_A}{CPI_B \times IC_B} = \frac{1.1 \times 10^9}{1.25 \times 1.2 \times 10^9} = 0.7333$$

Hence, the clock of the processor running compiler A's code is actually **slower** than the clock of the processor running compiler B's code.

**C)**

$$S = \frac{performance_{new}}{performance_{old}} = \frac{T_{exec-old}}{T_{exec-new}} = \frac{IC_{old} \times CPI_{old} \times T_{cycle-old}}{IC_{new} \times CPI_{new} \times T_{cycle-new}}$$

In this case, since both runs are on the same processor,

$$T_{cycle-old} = T_{cycle-new}$$

Hence,

$$S = \frac{IC_{old} \times CPI_{old}}{IC_{new} \times CPI_{new}}$$

Thus, compared to compiler A's code,

$$S = \frac{10^9 \times 1.1}{6 \times 10^8 \times 1.1} = 1.67$$

Compared to compiler B's code,

$$S = \frac{1.2 \times 10^9 \times 1.25}{6 \times 10^8 \times 1.1} = 2.27$$

**Problem (2)**

The execution time without hand-coding is:

$$T_{orig} = IC_{orig} \times CPI_{orig} \times T_{cycle}$$

The execution time with hand coding is:

$$T_{hand} = IC_{hand} \times CPI_{hand} \times T_{cycle}$$

The performance improvement (speedup) from hand coding is:

$$S = \frac{Perf_{hand}}{Perf_{orig}} = \frac{1/T_{hand}}{1/T_{orig}} = \frac{T_{orig}}{T_{hand}} = \frac{IC_{orig} \times CPI_{orig} \times T_{cycle}}{IC_{hand} \times CPI_{hand} \times T_{cycle}} = \frac{IC_{orig} \times CPI_{orig}}{IC_{hand} \times CPI_{hand}}$$

Thus:

$$\frac{IC_{hand}}{IC_{orig}} = \frac{CPI_{orig}}{CPI_{hand}} \times \frac{1}{S}$$

The problem states that:

$$CPI_{hand} = \frac{CPI_{orig}}{2.3} \quad \text{and} \quad S = 9.7$$

Hence,

$$\frac{IC_{hand}}{IC_{orig}} = \frac{2.3}{9.7} = 0.237$$

The instruction count is reduced to 23.7% of the original instruction count, i.e., there is a 76.3% reduction in the instruction count.

**Problem (3)**

$$\text{Time} = \#\text{instruction} \times \text{CPI} \times \frac{1}{\text{Clock rate}}$$

$$\#\text{instruction} = \frac{\text{Time} \times \text{Clock rate}}{\text{CPI}}$$

$$\text{CPI} = \frac{\text{Time} \times \text{Clock rate}}{\#\text{instruction}}$$

$$\#\text{cycle} = \text{CPI} \times \text{instruction} = \text{Time} \times \text{Clock rate}$$

For Program $P_A$,

$$\#\text{instruction} = \frac{3.8 \times \text{Clock rate}}{2.9}$$

$$\#\text{cycle} = 3.8 \times \text{Clock rate}$$

For Program $P_B$,

$$\#\text{instruction} = \frac{8.5 \times \text{Clock rate}}{3.7}$$

$$\#\text{cycle} = 8.5 \times \text{Clock rate}$$

The typical workload has

$$\#\text{instruction} = \frac{3.8' \text{ Clock rate}}{2.9} + \frac{8.5' \text{ Clock rate}}{3.7}$$

$$\#\text{cycle} = 3.8' \text{ Clock rate} + 8.5' \text{ Clock rate}$$

$$\text{CPI} = \frac{\#\text{cycle of workload}}{\#\text{instruction of workload}}$$

$$= \frac{(3.8+8.5)' \text{ Clock rate}}{\frac{3.8' \text{ Clock rate}}{2.9} + \frac{8.5' \text{ Clock rate}}{3.7}}$$

$$= \frac{12.3}{\left(\frac{3.8}{2.9} + \frac{8.5}{3.7}\right)}$$

$$= 3.409$$

**Problem (4)**

A) $\text{CPI} = \sum_i F_i \times \text{CPI}_i$

$\text{CPI}_{\text{non-fp}} \times 0.77 + 5.0 \times 0.23 = 3.6$

$\text{CPI}_{\text{non-fp}} = 3.182$

After improving the design of the floating point ALU,

$\text{CPI}_{\text{overall}} = 3.182 \times 0.77 + 3 \times 0.23 = 3.14$

B) The execution time of a program is

$$T = IC \times CPI \times T_{cycle}$$

Hence, when the only difference is CPI:

$$\frac{T_{new}}{T_{old}} = \frac{IC \times CPI_{new} \times T_{cycle}}{IC \times CPI_{old} \times T_{cycle}} = \frac{CPI_{new}}{CPI_{old}}$$

$$T_{new} = T_{old} \times \frac{CPI_{new}}{CPI_{old}}$$

It will take $190 \times 3.14 / 3.6 = 165.7$ seconds.

**Problem (5)**

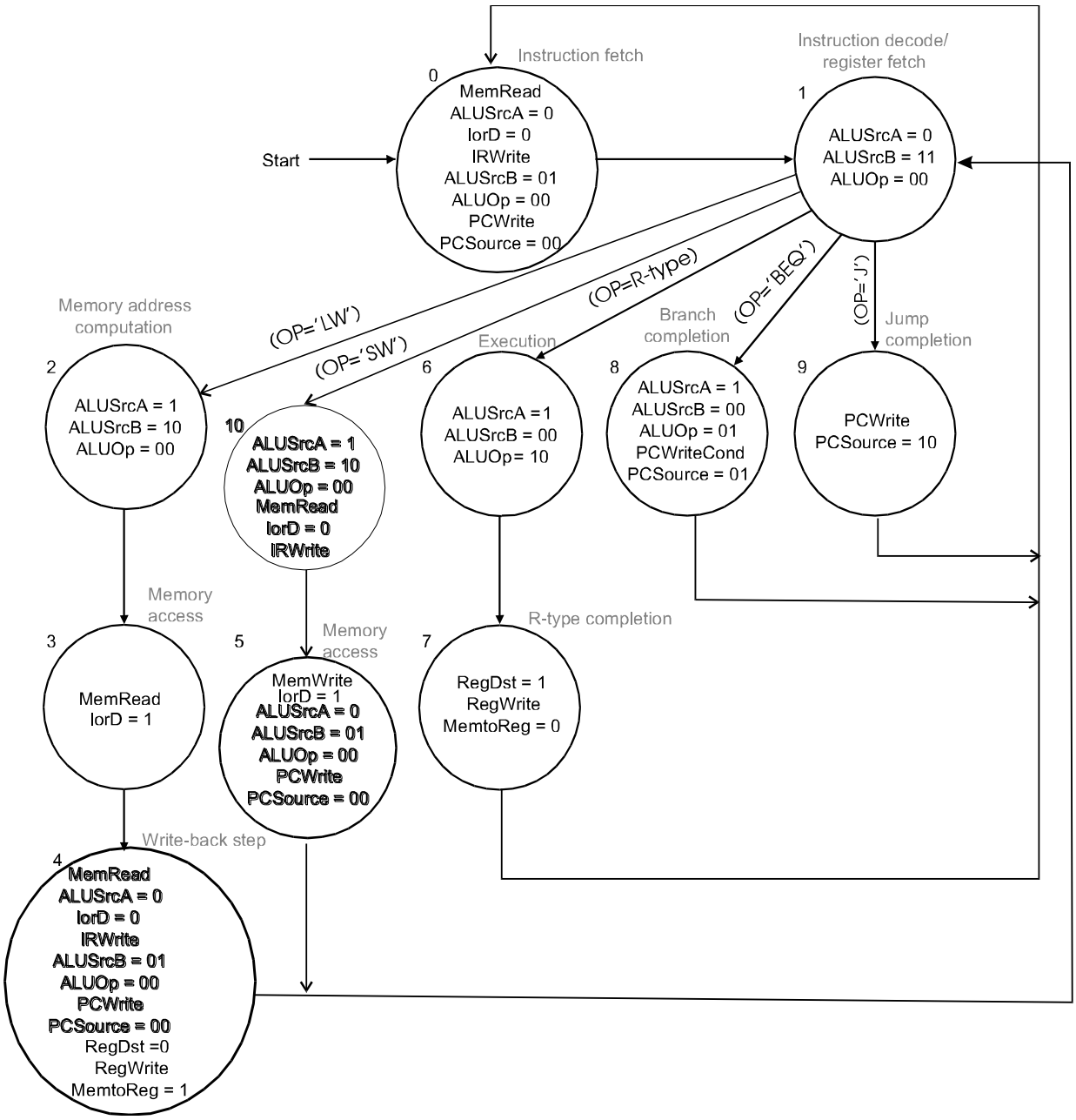
During the last cycle (state 5) of a `sw` instruction, the next instruction cannot be fetched since the memory is being used by the `sw` instruction. Hence, the next instruction must be fetched earlier. If the next instruction is fetched in state 2, the instruction register will be overwritten and the register number of the register into which the `lw` instruction needs to write the loaded data will be lost. Hence, in order to fetch the next instruction during the sw instruction, we must provide a separate "state 2" for the `lw` and `sw` instructions. For the `sw` instruction, we introduce a new state (state 10) during which the effective address is computed (as in state 2) **and** the next instruction is fetched.The PC is incremented in state 5 since ALU is used to compute the effective address in state 10. For the `lw` instruction, the last state (state 4) does not use the memory or the ALU so the next instruction is fetched and the PC is incremented during state 4. Since the next instruction will be at the output of the instruction register only after the end of state 4, this does not interfere with the proper execution of the register write for the `lw` instruction.

**A)** Introduce a new state, State 10, during which the effective address calculation for `sw` is performed. The orginal State 2 is only used for `lw`. For `lw`, the next instruction fetch and PC incrementation is performed in State 4, in parallel with writing the result to the register file. For `sw`, the next instruction fetch is performed in State 10, in parallel with the effective address calculation, while the PC incrementation is performed in State 5, in parallel with the memory write.

**B)** No modification to the datapath.

**C)** No new control signals.

**D)**



**Problem (6)**

In the non-pipelined (single-cycle) processor the latency of an `lw` instruction is a single clock cycle: 1250ps.
The latency of an `lw` instruction in a pipelined processor is the sum of the time through all stages. Therefore, the total latency is the product of the clock cycle time and the number of stages in the pipeline

(five): 5 × 350ps = 1750ps.

**Problem (7)**

In order to minimize clock cycle time, the longest stage in each case should be split.

Splitting the *ID* stage adds two new stages of 175 ps each and leaves the *MEM* stage as the new longest stage with a latency of 300 ps. Therefore, the new clock cycle time is 300 ps.

**Problem (8)**

The code segment is labeled as such:

```
i1:  or   R1, R2, R3
i2:  or   R2, R1, R4
i3:  or   R1, R1, R2
```

| Instruction | Dependent On | Dependence Type | Location |
|---|---|---|---|
| i2 | i1 | RAW | R1 |
| i2 | i1 | WAR | R2 |
| i3 | i1 | RAW | R1 |
| i3 | i1 | WAW | R1 |
| i3 | i2 | RAW | R2 |
| i3 | i2 | WAR | R1 |

**Problem (9)**

With the simple five-stage pipeline, registers are read in the order that the corresponding instructions appear in the program (ignoring branches). For each instruction, registers are read before any registers are written. Hence, it is guaranteed that if instruction X is fetched before instruction Y, all register reads by X will be done before any register write by Y. Hence, WAR dependencies through registers do not cause hazards.

With the simple five-stage pipeline, any register writes are done in the order that the corresponding instructions appear in the program (ignoring branches). Hence, WAW dependencies through registers do not cause hazards.

With the simple five-stage pipeline, memory reads and writes are performed in the order that the corresponding instructions appear in the program (ignoring branches). Hence, RAW, WAR, and WAW dependencies through memory do not cause hazards.

In conclusion, with the simple five-stage pipeline, only RAW dependencies through registers can cause hazards.

Consider the RAW dependency from i1 to i2.

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 |
|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | |
| i2 | or | | IF | ID | EX | MEM | WB |

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 |
|---|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | | |
| | … | | | | | | | | |
| i2 | or | | | | IF | ID | EX | MEM | WB |

Since there is no forwarding, the earliest that the i2, `or` instruction can read R1 from the register file is during the same cycle (CC5) that the i1, `or` instruction writes R1 to the register file. This dependency is a hazard.

Consider the RAW dependency from i2 to i3.

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 |
|---|---|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | | | |
| | … | | | | | | | | | |
| i2 | or | | | | IF | ID | EX | MEM | WB | |
| i3 | or | | | | | IF | ID | EX | MEM | WB |

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 | CC11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | | | | | |
| | … | | | | | | | | | | | |
| i2 | or | | | | IF | ID | EX | MEM | WB | | | |
| | .... | | | | | | | | | | | |
| i3 | or | | | | | | | IF | ID | EX | MEM | WB |

Since there is no forwarding, the earliest that the i3 `or` instruction can read R2 to the register file is during the same cycle (CC8) that the i2 `or` instruction writes R2 to the register file. This dependency is a hazard.

These hazards are eliminated by inserting two `nop` instructions in between i1 and i2 and `nop` instructions in between i2 and i3.

```
i1:  or   R1, R2, R3
     nop
     nop
i2:  or   R2, R1, R4
     nop
     nop
i3:  or   R1, R1, R2
```

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 | CC8 | CC9 | CC10 | CC11 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | | | | | |
| | nop | | IF | ID | EX | MEM | WB | | | | | |
| | nop | | | IF | ID | EX | MEM | WB | | | | |
| i2 | or | | | | IF | ID | EX | MEM | WB | | | |
| | nop | | | | | IF | ID | EX | MEM | WB | | |
| | nop | | | | | | IF | ID | EX | MEM | WB | |
| i3 | or | | | | | | | IF | ID | EX | MEM | WB |

**Problem (10)**

With the simple five-stage pipeline, only RAW dependencies through registers can cause hazards.

Consider the RAW dependency from i1 to i2.

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 |
|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | |
| i2 | or | | IF | ID | EX | MEM | WB |

The result of the i1 `or` instruction is available at the beginning of its MEM stage (CC4). This value is needed by the i2 `or` instruction at the beginning of its EX stage (CC4). This result can be forwarded from the EX/MEM register to the ALU inputs during CC4. Thus, this dependency does not cause a hazard.

Consider the RAW dependency from i2 to i4.

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC& |
|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | |
| i2 | or | | IF | ID | EX | MEM | WB | |
| i3 | or | | | IF | ID | EX | MEM | WB |

The result of the i2 `or` instruction is available at the beginning of its MEM stage (CC5). This value is needed by the i3 `or` instruction at the beginning of its EX stage (CC5). This result can be forwarded from the EX/MEM register to the ALU inputs during CC4. Thus, this dependency does not cause a hazard.

Since there are no hazards due to these dependencies, the code remains the same.

```
i1:  or   R1, R2, R3
i2:  or   R2, R1, R4
i3:  or   R1, R1, R2
```

| | | CC1 | CC2 | CC3 | CC4 | CC5 | CC6 | CC7 |
|---|---|---|---|---|---|---|---|---|
| i1 | or | IF | ID | EX | MEM | WB | | |
| i2 | or | | IF | ID | EX | MEM | WB | |
| i3 | or | | | IF | ID | EX | MEM | WB |

**Problem (11)**

There are no signals asserted during the IF and ID stages.

| | Instruction | EX Stage | MEM Stage | WB Stage |
|---|---|---|---|---|
| A) | `sw $14, 44($3)` | ALUSrc = 1<br>ALUOp = 00 | Branch = 0<br>MemWrite = 1<br>MemRead = 0 | |
| B) | `sub $21, $9, $2` | ALUSrc = 0<br>ALUOp = 10<br>RegDst = 1 | Branch = 0<br>MemWrite = 0<br>MemRead = 0 | MemtoReg = 0<br>RegWrite = 1 |

# Solutions to Homework #4 Practice Problems

**Problem (12)**

Denote the clock rate (frequency) by F. Denote the program execution time in seconds as $T_{prog}$.

Everything is derived directly from the basic performance equation:

$$T_{prog} = IC \times CPI \times T_{cycle} = IC \times CPI \times \frac{1}{F}$$

a. $$Performance = \frac{\#instructions}{\#seconds} = \frac{\#instructions}{\#cycles} \times \frac{\#cycles}{\#seconds} = \frac{1}{CPI} \times F = \frac{F}{CPI}$$

Thus, the performance of P1, P2, P3 is:

$Performance_{P1} = 3 \times 10^9 / 1.5 = 2 \times 10^9$ instructions per second

$Performance_{P2} = 2.5 \times 10^9 / 1.0 = 2.5 \times 10^9$ instructions per second

$Performance_{P3} = 4.0 \times 10^9 / 2.2 = 1.81 \times 10^9$ instructions per second

Hence, processor P2 has the highest performance expressed in instruction per second.

b. $$\#cycles = \frac{\#cycles}{\#seconds} \times T_{prog} = F \times T_{prog}$$

Thus, the number of cycles of P1, P2, P3 is:

$Clock\ cycles_{P1} = 3 \times 10^9 \times 10 = 3 \times 10^{10}$ cycles

$Clock\ cycles_{P2} = 2.5 \times 10^9 \times 10 = 2.5 \times 10^{10}$ cycles

$Clock\ cycles_{P3} = 4.0 \times 10^9 \times 10 = 4.0 \times 10^{10}$ cycles

$$InstructionCount(IC) = \frac{T_{prog}}{CPI \times T_{cycle}} = \frac{T_{prog} \times F}{CPI}$$

Thus, the number of instructions of P1, P2, P3 is:

$IC_{P1} = 10 \times 3 \times 10^9 / 1.5 = 2 \times 10^{10}$ instructions

$IC_{P2} = 10 \times 2.5 \times 10^9 / 1.0 = 2.5 \times 10^{10}$ instructions

$IC_{P3} = 10 \times 4.0 \times 10^9 / 2.2 = 1.81 \times 10^{10}$ instructions

c. $F = \frac{IC \times CPI}{T_{prog}}$

Let $F^n$, $IC^n$, $CPI^n$ and $T_{sub}{}^n$ denote corresponding terms for the "new" system.

Hence, $T_{prog}{}^n = 0.7 \times T_{prog}$, $CPI^n = 1.2 \times CPI$ and $IC^n = IC$

Thus, $F_n = \frac{IC \times 1.2 \times CPI}{0.7 \times T_{prog}} = 1.71 \times \frac{IC \times CPI}{T_{prog}} = 1.71 \times F$

Therefore, the clock rate we should have to get that reduction is:

$1.71 \times 3 = 5.13$ GHz for P1

$1.71 \times 2.5 = 4.275$ GHz for P2

$1.71 \times 4.0 = 6.84$ GHz for P3

**Problem (13)**

$CPI = \sum_i F_i \times CPI_i$

CPI for MFP $= 0.1 \times 6 + 0.15 \times 4 + 0.05 \times 20 + 0.7 \times 2 = 3.6$

CPI for MNFP $= 2$

**Execution-rate = #instructions/second**

= #inst/cycle × #cycles/sec
= 1/(#cycles/inst) × #cycles/sec
= clock-rate/CPI

Execution-rate for MFP $= 1000 \times 10^6 / 3.6 = 278$ MIPS

Execution-rate for MNFP $= 1000 \times 10^6 / 2 = 500$ MIPS

**Problem (14)**

# of integer instructions = 300,000,000 × 70% = 210,000,000
# of floating-point multiply instructions = 300,000,000 × 10% = 30,000,000
# of floating-point add instructions = 300,000,000 × 15% = 45,000,000
# of floating-point divide instructions = 300,000,000 × 5% = 15,000,000

**For MNFP, total number of integer instructions =**

210,000,000 +
30,000,000 × 30 +
45,000,000 × 20 +
15,000,000 × 50

= 2,760,000,000 integer instructions

## Problem (15)

# of integer instructions = 5,000,000,000 × 70% = 3,500,000,000
# of floating-point multiply instructions = 5,000,000,000 × 10% = 500,000,000
# of floating-point add instructions = 5,000,000,000 × 15% = 750,000,000
# of floating-point divide instructions = 5,000,000,000 × 5% = 250,000,000

**For MNFP, total number of integer instructions =**

3,500,000,000 +
500,000,000 × 30 +
750,000,000 × 20 +
250,000,000 × 50
= 46,000,000,000 integer instructions

For MFP:

**Execution time = (3,500,000,000 × 2 + 500,000,000 × 6 +**
750,000,000 × 4 + 250,000,000 × 20) / 1000 MHz
= 18 seconds

For MNFP:

Execution time = 46,000,000,000 × 2 / 1000 MHz
= 92 seconds

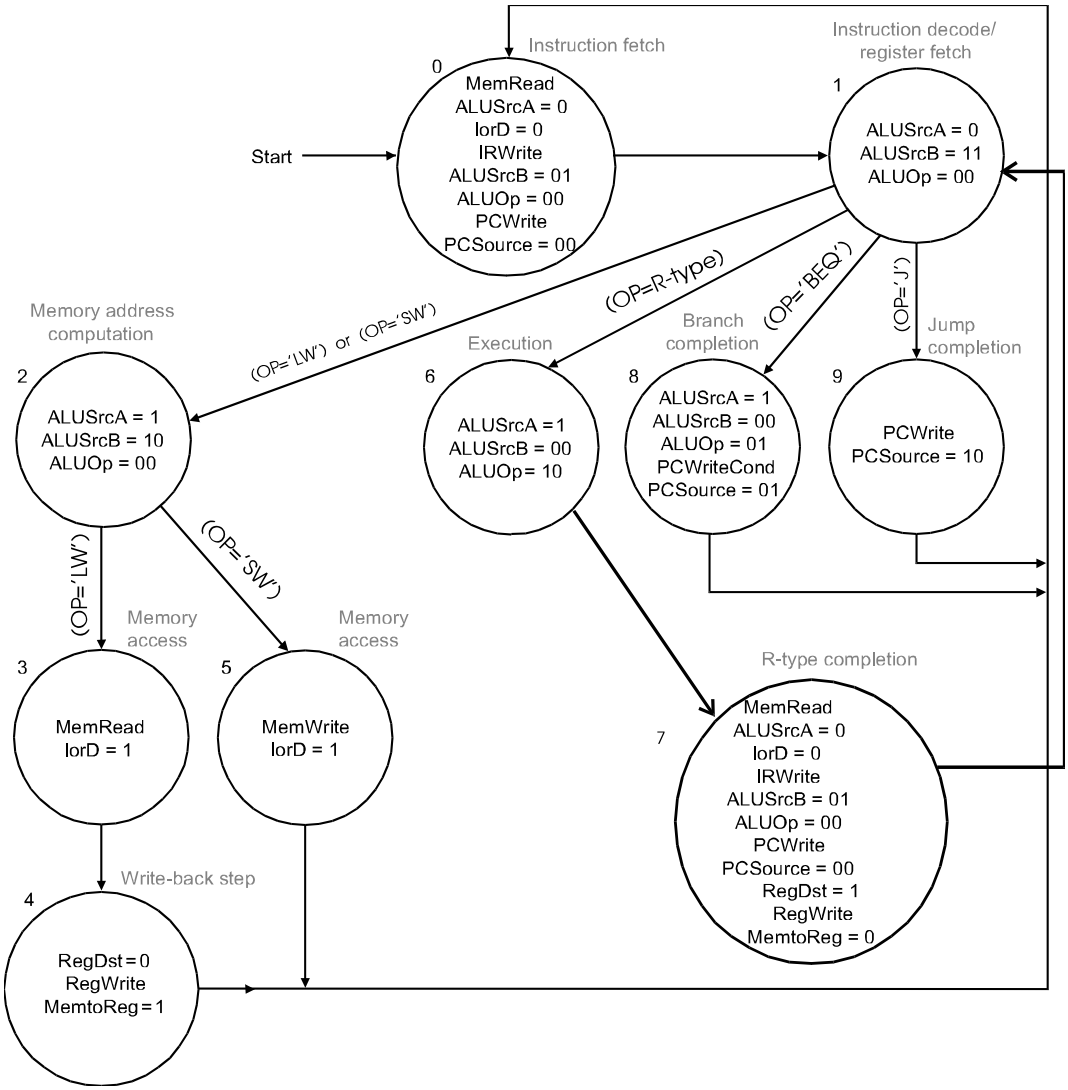
## Problem (16)

A) There are no resource conflicts among the three operations: instruction fetch, incrementing the PC, and writing to the register file. Thus, the instruction fetch and PC incrementation are performed during the last cycle of R-type instructions. This allows a transition from State 7 directly to State 1, skipping State 0. State 0 is still needed for instructions that do not follow R-type instructions.

B) No modification to the datapath

C) No new control signals

D)




**Problem (17)**

The clock cycle time for the non-pipelined (single-cycle) processor is the latency of the longest instruction, `lw`, which requires all stages:

250ps + 350ps + 150ps + 300ps + 200ps = 1250ps

The clock cycle time for the pipelined processor is the latency of the longest stage.

The longest stage is the *ID* stage with a latency of 350ps

**Problem (18)**

A)

The ALU operation now takes (1 – 0.25) × 200 ps = 150 ps. The cycle time of the pipelined implementation does not change because the instruction fetch and data access stages still take 200 ps each. The cycle time of the single-cycle implementation is the time it takes to execute the slowest instruction, `lw`: 200 ps + 100 ps + 150 ps + 200 ps + 100 ps = 750 ps.

For a program consisting of IC instructions:

$T_{single} = IC \times 750 \text{ ps}$

$T_{pipe} = (5 + IC - 1) * 200 \text{ ps} = 800 \text{ ps} + IC \times 200 \text{ ps}$

$$\text{Speedup} = \frac{T_{single}}{T_{pipe}} = \frac{IC \times 750 \text{ ps}}{800 \text{ ps} + IC \times 200 \text{ ps}} \approx \frac{IC \times 750 \text{ ps}}{IC \times 200 \text{ ps}} = 3.75$$

B)

The ALU operation now takes 25% more time, or 1.25 × 200 ps = 250 ps. The cycle time of the single-cycle implementation is the time it takes to execute the slowest instruction, `lw`: 200 ps + 100 ps + 250 ps + 200 ps + 100 ps = 850 ps. The cycle time of the pipelined implementation changes to 250 ps in order to accommodate the slowest stage, ALU operation.

For a program consisting of IC instructions:

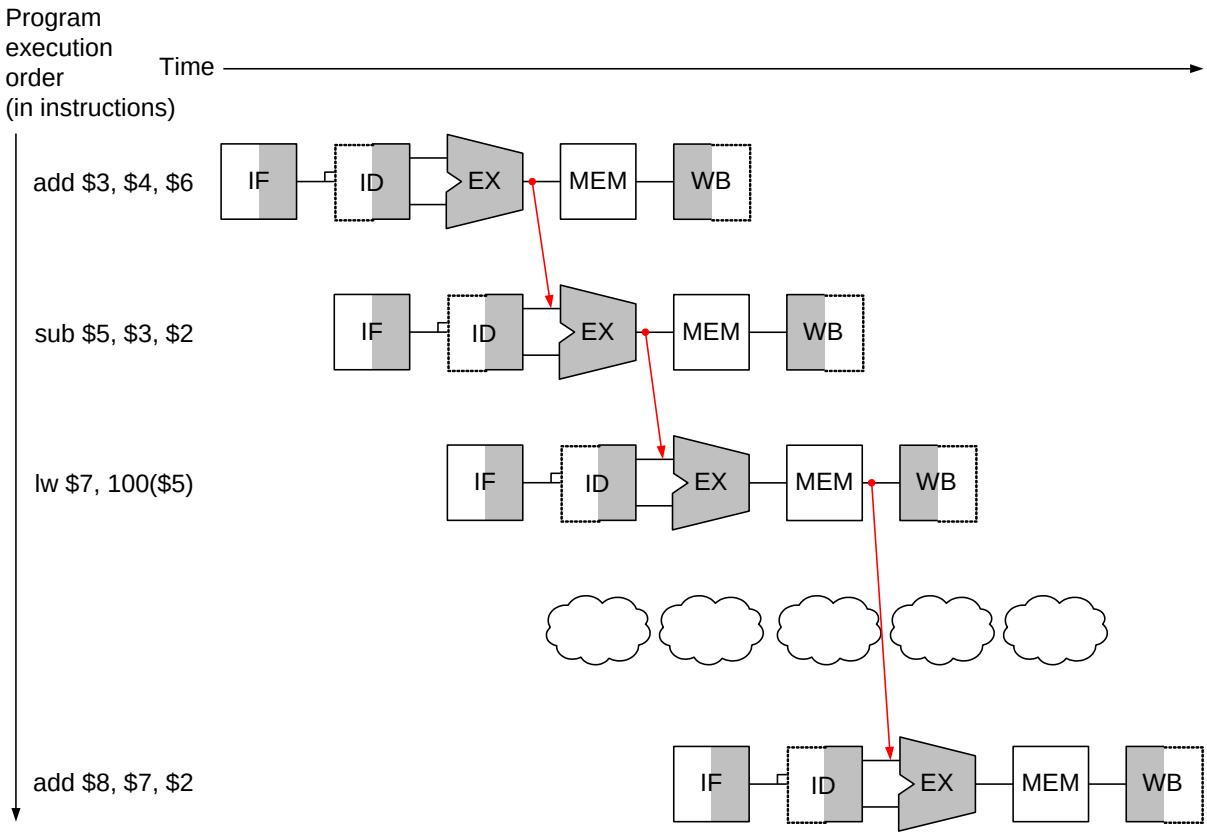
$T_{single} = IC \times 850 \text{ ps}$

$T_{pipe} = (5 + IC - 1) * 250 \text{ ps} = 1000 \text{ ps} + IC \times 250 \text{ ps}$

$$\text{Speedup} = \frac{T_{single}}{T_{pipe}} = \frac{IC \times 850 \text{ ps}}{1000 \text{ ps} + IC \times 250 \text{ ps}} \approx \frac{IC \times 850 \text{ ps}}{IC \times 250 \text{ ps}} = 3.4$$

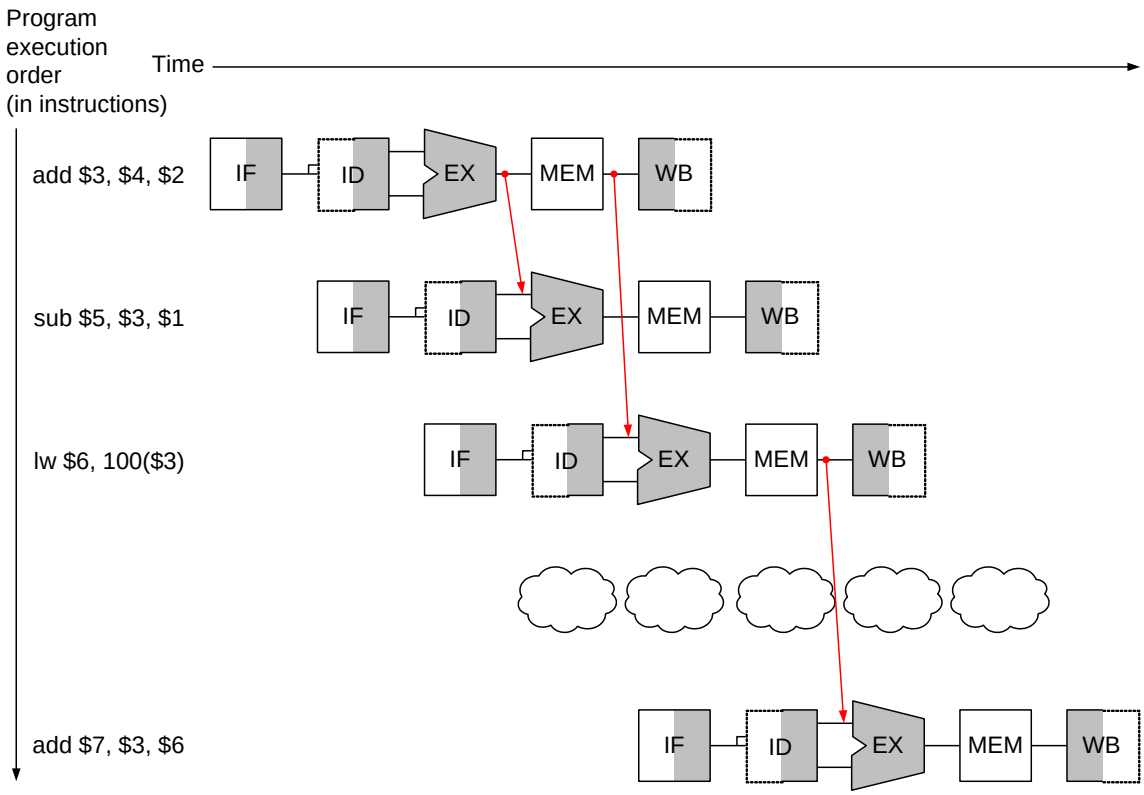
**Problem (19)**

A forwarding path from the EX result to the EX input is needed for the dependency between the first `add` and `sub` instructions and for the dependency between the `sub` and `lw` instructions. A forwarding path is needed between the MEM output and the EX input for the dependency between `lw` and second `add` instructions.




**Problem (20)**

The second instruction depends on the first ($3); this data hazard is solved by forwarding.
The third instruction depends on the first ($3); this data hazard is solved by forwarding.
The fourth instruction depends on the third ($6); this data hazard is solved by a stall and forwarding.
The fourth instruction also depends on the first ($3), but there is no data hazard.




**Problem (21)**

The first *lw* instruction requires 5 cycles to execute. Since the datapath is pipelined, each subsequent instruction without hazards can complete in 1 cycle. There are 2 data dependencies in this problem: 1) the *add* instruction depends on the *lw* that immediately precedes it, 2) the *lw* instruction depends on the *add* that immediately precedes it.

1) Since there is data forwarding, case (2) does not require a pipeline stall. However, in case (1), since the *add* instruction depends on the previous *lw* instruction, there is 1 stall cycle for each *add* instruction.

   CPI = # of cycles / # of instructions

   = ((# cycles for first *lw*) + (# cycles for *add*) + (# cycles for *lw*)) / # of instr.

   = $(5 + 500 * 2 + 499) / 10^3$

   = $1504 / 10^3$

   = 1.504

2) Without data forwarding, each subsequent instruction after the first *lw* requires 2 stall cycles. Hence, each subsequent instruction requires 3 cycles to complete.

   CPI = $(5 + (10^3 - 1) * 3) / 10^3$

   = 3.002