CPSC 3300 Midterm Study Guide

Brendan McGuire (bmmcgui@clemson.edu), Patrick Smathers (psmathe@clemson.edu), Justin Kristensen (jkriste@clemson.edu)

Disclaimer: This is an intentionally abridged version of course content focused on material seen in example questions for the exam. Please consult Faith's Notes for a complete reference of course content.

Disclaimer: I try to be accurate, but I am a student like you! If you see anything inaccurate or misleading, please let me know!

Introductory Material

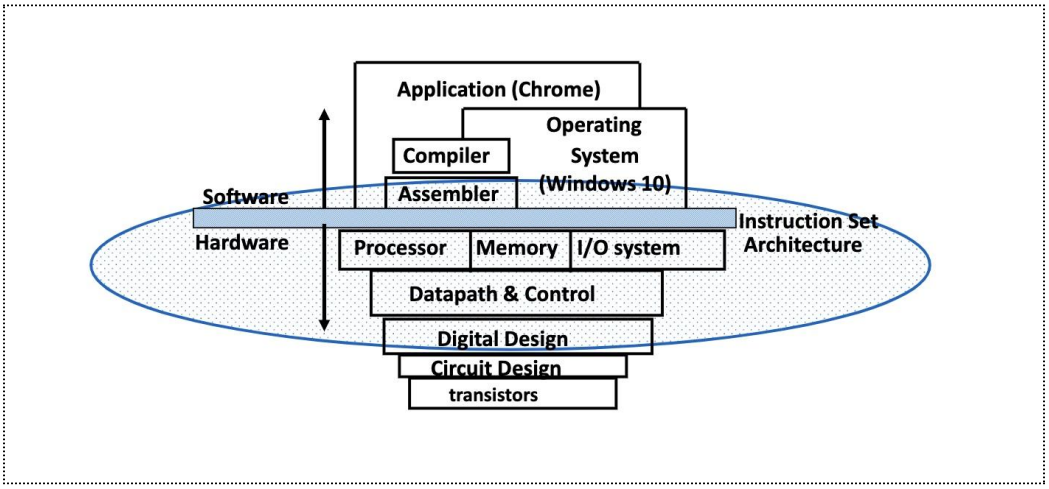
Traditionally, computers are divided into one of a couple of classes based on their use.

- ➔ **Personal Computers** are designed to meet the needs of a small number of people (like a family)
- ➔ **Servers** that provide services (compute, storage, access to hardware) to many computers across a network
- ➔ **Embedded Computers** are typically low-power, single-function computers that reside within a specific device.

More recently, new technology has blurred the lines between these traditional classes. Personal computers have largely been supplemented or replaced by smartphones. Servers are often abstracted away from their users through Infrastructure as a Service (AWS), Platform as a Service (GitHub), and Software as a Service (Figma) offerings.

Layers of Abstraction

This course focuses on the layer of abstraction represented in blue here, on the border of the software and hardware sections of the diagram.



Measuring Performance

When measuring performance in a computer system, we often divide the measurement into two different aspects: (1) **elapsed time**, which measures the total time spent on a specific task, and (2) **CPU time**, which is composed of all of the time spent executing a specific task. Elapsed time is more all-encompassing, including waiting for I/O, executing other processes, or in the kernel.

Powers of 10

- exa: 10^{18}
- peta: 10^{15}
- tera: 10^{12}
- giga: 10^{9}
- milli: 10^{-3}
- micro: 10^{-6}
- nano: 10^{-9}
- pico: 10^{-12}

Example: Matching

Words:
Server, Embedded Device, Transition, Moore's Law, Dennard Scaling, Speedup, Throughput, CPU / Execution Time, Benchmark, SPEC

1. A computer that provides computation, file storage, and/or printing to multiple users on a network. **Server**. Distinguished from an embedded computer by providing services over the network
2. A computer used inside another device running one or more predetermined apps. **Embedded Device.**
3. A semiconductor device is used as a switch. **Transistor**
4. The observation is that the number of transistors in a dense circuit doubles approx. every two years. **Moore's Law.**
5. As transistors get smaller, their power density stays constant. **Dennard Scaling**
6. The ratio of execution times of two computer systems. **Speedup**
7. Measure of work done per unit of time. **Throughput**
8. Time spent processing a given job. **Execution Time (or CPU Time)**
9. A test that measures the performance of a system or subsystem on a well-defined task. **Benchmark** s
10. A set of benchmark suites is used to compare the performance of various desktops and servers. **SPEC**. The Standard Performance Evaluation Cooperative designs benchmarks for various aspects of design

Example: Execution Time
Find the execution time for a program that executed 60 million instructions on a processor with an average CPI of 1.5 and a clock frequency of 2 GHz.

Total Clock Cycles: 60 million instructions * 1.5 CPI = 90 million clock cycles
Execution Time: $90 * 10^6$ clock cycles / $2 * 10^9$ cycles per second = 0.045 seconds

Example: CPI

A programmer writes a program that compiles down to have exactly 60 million instructions. The distribution and cycle count of each type of instruction are shown in the table below. If the program executes on a single-issue core with a clock rate of 2.4 Ghz, find the execution time.

```
CPI = (0.6)(1) + (0.2)(4) + (0.2)(2)
CPI = 1.8

t = (60 * 10^6) * (1.8) / (2.4 * 10^9)

t = 0.045 seconds
```

Type	Frequency	Cycles
ALU	0.6	1
ld/st	0.2	4
branch	0.2	2

Now, consider a new version of a compiler, which reduces the total number of instructions by 10% (i.e., now only 90% of instructions), and alters the distributions of instructions (now shown below). Compute the total speed up.

```
CPI = (0.8)(1) + (0.1)(4) + (0.1)(2)
CPI = 1.4

instructions = 0.9 * 60 = 54

t = (54 * 10^6) * (1.4) / (2.4 * 10^9)

t = 0.0315 seconds
```

Type	Frequency	Cycles
ALU	0.8	1
ld/st	0.1	4
branch	0.1	2

```
speedup =  0.045 / 0.0315 = 1.42x
```

Logic Functions

An abstraction we use to represent logic circuits. We can precisely specify a logic function using truth tables containing every combination of inputs and their corresponding output.

A	B	Output
0	0	1
0	1	1
1	0	0
1	1	0

More precisely, if there are n inputs (each of which can either be true or false), then there must be exactly 2^n rows in the truth table to account for all of the combinations of the inputs.

Boolean Equations

Alternatively, we can specify logic functions using boolean equations. These equations can have multiple inputs but have a single output. We utilize the + symbol to represent logical OR, “multiplication” to represent logical AND, and a ‘ to represent logical NOT

Expression	What It Means
`ab + c`	`(a AND b) OR c`
`a’`	`NOT a`
`a’ + b’`	`NOT a OR NOT b`
`a’bc + b’c’`	`(NOT a AND b AND c) OR (NOT b AND NOT c)`

Sum of Products

One particularly useful form of boolean expressions is the Sum of Products, which is exactly what it sounds like.

$$F(a, b, c) = ab + bc$$

Sum of Minterms

A canonical form for a boolean expression where *every term* of the expression is a **minterm.** In this context, minterm means that each term in the expression (separated by OR) contains every variable in the system. For example, the following expression is a **sum of minterms** because each term (abc, a’bc, and ab’c) includes all variables in the system (or negations) at least once.

$$F(a, b, c) = abc + a'bc + ab'c$$

Boolean expressions in the sum of minterms form are extremely easy to convert to an electrical circuit with a simple 2-level design.

Truth Table to Sum of Minterms

We can convert a truth table to a boolean expression by considering all of the rows where the truth table's output is 1. For each row where the output is 1, add a term to the boolean expression

Ex: Convert the truth table below to a boolean expression in sum-of-minterms form

A	B	C	F(A,B,C)
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

This truth table is logically equivalent to

F(A,B,C) = a'b'c + a'bc + ab'c

Sum of Products to Sum of Minterms

While a sum of minterms is a sum of products, not all terms in a sum of products have every variable in the system. If the variable v is missing from a term, we can multiply the term by the identity v + v' (v OR NOT v) to convert the term into 2 terms, each of which now contains the missing variable.

Example: Convert the following sum of products to sums of minterms.

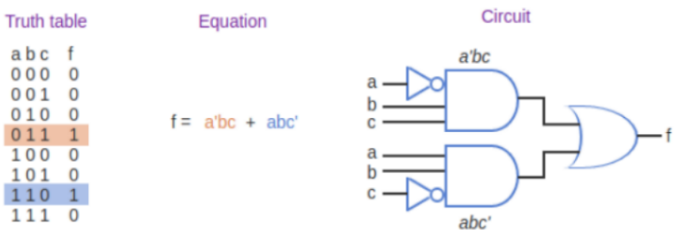
f(a, b, c) = ab + bc

f(a, b, c) = ab(c + c') + (a + a')bc

= abc + abc' + abc + a'bc

Sum of Minterms to Circuit

You can easily convert a sum of minterms into a circuit with a 2-level design.



Example: Sum of Minterms

Given the following table of inputs A, B, C and output F, write the Sum of Minterms.

A	B	C	F(A, B, C)
0	0	0	1
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	1
1	0	1	1
1	1	0	0
1	1	1	1

F = *fn(A, B, C)* = A'B'C' + A'BC + A'BC' + A'BC + AB'C' + ABC

A	B	C	F(A, B, C)
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	1
1	1	1	0

F = *fn(A, B, C)* = A'BC' + AB'C' + ABC'

Combinational & Sequential Logic

A combinational logic circuit's output depends only on the present combination of input values.

➔ For example: consider a simple adder circuit. It takes 2 bits (A and B), and has 2 outputs, but maintains no internal state. The output is purely a function of the 2 external inputs A and B

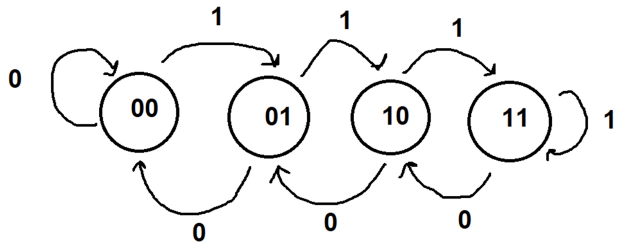
A sequential logic circuit maintains at least one bit of state, and the current value O(t) is used as input. This allows us to implement **finite state machines**, and any system that has to maintain some level of internal state.

➔ For example: a 1-bit branch predictor that maintains a single bit of state (whether the branch previously jumped or not). The status of the state bit O(t) is used to predict the next jump O(t+1)

Example: Counter

Consider a two-bit saturating up down counter with input I. When I is 1, the count is incremented (00 → 01), saturating at state 11. When I is 0, the count is decremented, saturating at 00.

1) Draw the state transition diagram. Label the up transitions with 1 and the down transitions with 0.



Explanation: Because there are 2 bits of state there are 4 possible states for the counter, holding either 00, 01, 10, or 11. This is a sequential logic circuit. Depending on the value of I, the state transitions as you would expect, saturating at 11 for increment and 00 for decrement.

2) Give a truth table for each bit of state, $Q_0(t)$ and $Q_1(t)$

This is a sequential logic circuit, with the outputs $Q_0(t+1)$ and $Q_1(t+1)$ dependant on the value of I, $Q_0(t)$ and $Q_1(t)$. Note that Q_0 represents the lower bit of state and Q_1 represents the high bit. Follow the state transitions above to find the values of the truth table.

I	$Q_1(t)$	$Q_0(t)$	$Q_1(t+1)$	$Q_0(t+1)$
0	0	0	0	0

1	0	0	0	1
0	0	1	0	0
1	0	1	1	0
0	1	0	0	1
1	1	0	1	1
0	1	1	1	0
1	1	1	1	1

3) Give simplified logic expressions for $Q_1(t + 1)$ and $Q_0(t + 1)$

Convert the truth table into sum of minterms like above by considering the rows where the output we care about is 1.

$Q_0(t + 1) = IQ_1(t)'Q_0(t)' + I'Q_1(t)Q_0(t)' + IQ_1(t)Q_0(t)' + IQ_1(t)Q_0(t)$

$Q_1(t + 1) = I'Q_1(t)Q_0(t)' + IQ_1(t)Q_0(t)' + I'Q_1(t)Q_0(t) + IQ_1(t)Q_0(t)$

Simplify using properties of boolean algebra, and identify identities

$Q_1(t + 1) = IQ_1(t)'Q_0(t) + I'Q_1(t)Q_0(t)' + IQ_1(t)Q_0(t)' + IQ_1(t)Q_0(t)$

$= IQ_1(t)'Q_0(t) + (I' + I)Q_1(t)Q_0(t)' + IQ_1(t)Q_0(t)$

$= \mathbf{IQ_1(t)'Q_0(t) + Q_1(t)Q_0(t)' + IQ_1(t)Q_0(t)}$

$Q_0(t + 1) = I'Q_1(t)Q_0(t)' + IQ_1(t)'Q_0(t) + (I' + I)Q_1(t)Q_0(t)$

$= \mathbf{I'Q_1(t)Q_0(t)' + IQ_1(t)'Q_0(t) + Q_1(t)Q_0(t)}$

MIPS Instructions

There are many different instruction set architectures (ISAs), but this course will focus primarily on MIPS, a RISC instruction set that is relatively simple.

Types of Instructions

All instructions in MIPS are 32 bits long and divided into three formats. The format of an instruction is distinguished by its opcode.

- ➔ **R Type** Instructions (ex: add, or, slt) have an opcode of 0 and use the funct field to distinguish between the different operations
- ➔ **J Type** instructions (ex: j, jal) have an opcode of 2 or 3, and just have an address. Jump instructions use pseudo-absolute addressing, in which the upper 4 bits of the computed address are taken relatively from the program counter. The lower 2 bits are assumed to be 0 (since all PC values are multiples of 4).
- ➔ **I Type** instructions (ex: beq, lw, sw) instructions feature a 16-bit immediate, usually sign extended to 32 bits.

MIPS Instruction Formats						
R Type	opcode (6) **0**	rs (5 bits)	rt (5 bits)	rd (5 bits)	shamt (5 bits)	funct (6 bits)
J Type	opcode (6) **2,3**	address (26 bits)				
I Type	opcode (6) **>3**	rs (5)	rt (5)	immediate (16 bits)		

Data Pipeline

When executing instructions, we can divide the execution pipeline into one of 5 stages. Doing this will allow us to pipeline multiple instructions (i.e., execute instructions in different stages simultaneously).

Simple Five-Stage Pipeline

- ➔ Instruction Fetch (IF) – Read the instruction at the Program Counter
- ➔ Instruction Decode (ID) – Translate opcode into control signals and read registers
- ➔ Execution (EX) – Perform ALU operation, compute effective memory address
- ➔ Memory Access (MEM) – Access memory if needed
- ➔ Write Back (WB) – Update register file, write back to registers

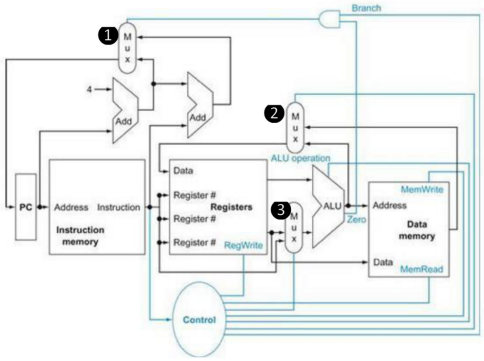
After each instruction phase, we write the results of that stage to a specific *pipeline register,* which allows us to suspend operations midway through execution, allowing for pipelining.

Generally speaking, RISC architectures are easier to pipeline than CISC because (1) instructions are typically only a couple of different formats, (2) operations on data apply only to data in registers, and the only operations that act on memory are load and store.

Control Signals

As part of the ID phase of the pipeline, the current instruction is fetched from the Program Counter, and decoded to see what parts of the CPU it will need to use. This is output via a number of **control signals** that will control the behavior of the CPU

1) Branch – Whether the instruction is a branch
2) MemRead – Whether memory is read from this instruction
3) MemWrite – Whether the instruction writes to memory
4) RegWrite – Whether the instruction writes to a register
5) Mux1 (How To Increment PC) – If this is a branch instruction, and the ALU is 0, than it takes the higher input (set PC to branch location). Else it takes the lower input (increment PC by 4)



6) Mux2 (Data Source) – If the instruction writes from memory → register, take the higher input. Else it takes the lower input (writes to register from ALU)
7) Mux3 (Data Into ALU) – If the instruction takes an immediate in the ALU, take the lower input. If the instruction takes a register value into the ALU, take the higher input
8) ALU Operation – The ALU operation (add, sub, and, or, etc.)

Example: Control Signals

Consider the instructions and give the output of each control signal in the CPU.

```
and R1, R2, R3 //Reg[1]<- Reg[2]&Reg[3]
```

1. Branch = **0** (no branch)
2. MemRead = **0** (does not read from memory)
3. MemWrite = **0** (does not write to memory)
4. RegWrite = **1** (writes to a register)
5. Mux1 = **lower** (PC incremented by 4, no branch)
6. Mux2 = **lower** (data comes from register, not memory)
7. Mux3 = **higher** (data comes from a register value into the ALU)
8. ALU Operation = **and**

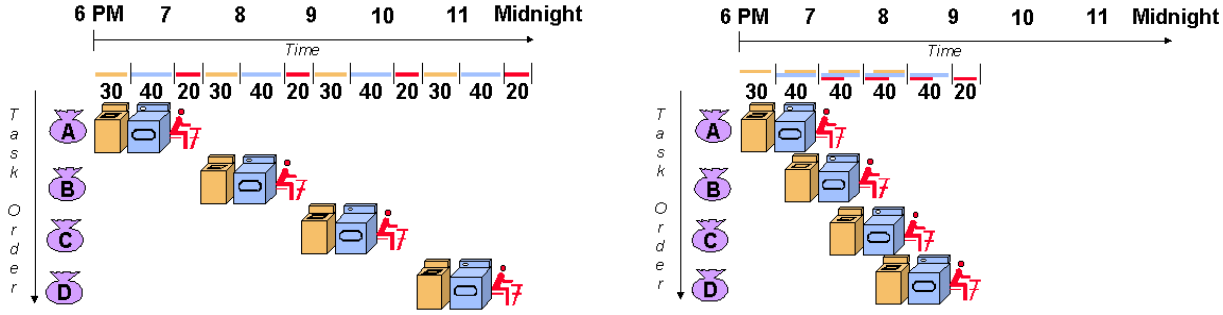
CPU Optimizations

Pipelined Instructions

We discussed above in Data Pipeline how the execution of instructions can be broken down into five stages.

1. Instruction Fetch (IF)
2. Instruction Decode (ID)
3. ALU Execution or Effective Memory Address Computation (EXE)
4. Memory Access (MEM)
5. Write Back to Main Registers (WB)

Because each stage in the pipeline utilizes different parts of the CPU, we can *execute multiple* instructions in parallel as long as they are in different stages. Pipelining is an implementation technique whereby multiple instructions are overlapped in execution



For a good metaphor, imagine the pipeline required to do your laundry. You must wash your clothes, dry them, and then put them away. While you only have one washer, dryer, and set of hands each to perform that stage in the pipeline, you can process multiple loads of laundry simultaneously by overlapping the execution.

Example: Pipelines

Identify the five stages of the simple pipeline we studied, and explain what each stage does when processing the instruction below.

```
and R1, R2, R3 //Reg[1]<- Reg[2]&Reg[3]
```

1. Instruction Fetch (IF) – Loads the and instruction from the PC
2. Instruction Decode (ID) – Decodes the instruction into a number of control signals (see Example: Control Signals), and loads R2 and R3 into the ALU
3. ALU Execution or Effective Memory Address Computation (EXE) – Executes the and operation
4. Memory Access (MEM) – Not utilized, no memory reads or writes
5. Write Back to Main Registers (WB) – The result from the ALU written into R1

Hazards & Data Dependence When Pipelining

The following are three main types of hazards and are situations that will prevent the next instruction from executing during its designated clock cycle when pipelining.

Structural Hazards

This type of hazard occurs when the hardware cannot support the combination of instructions we want to execute in the same clock cycle when pipelining.

Control Hazards

This type of hazard occurs when the pipeline makes incorrect branch prediction decisions. This results in instructions entering the pipeline that need to be discarded.

Data Hazards

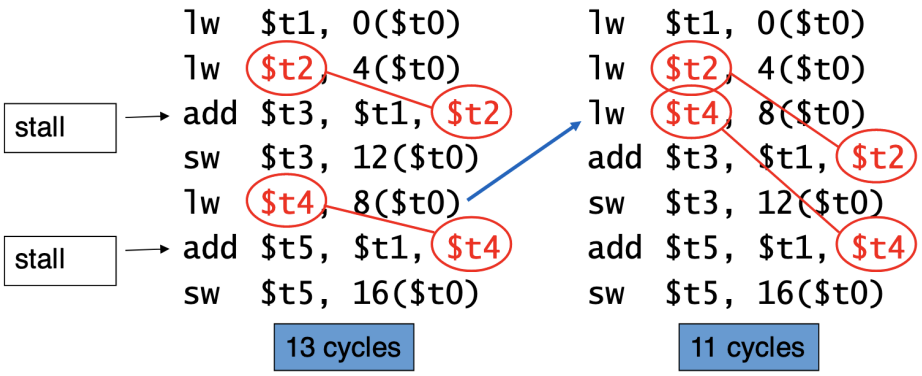
This type of hazard occurs when an instruction depends on the result of a previous instruction in a way that is exposed by the overlapping of instructions in the pipeline.

Read After Write (RAW)

A planned instruction cannot execute in the proper clock cycle since data that is needed to execute the instruction is not yet available.

Load-Use Data Hazard

A specific form of data hazard where the data being loaded by a load instruction has not yet become available when another instruction needs it. This type of data hazard needs to stall along with forwarding. In order to avoid these stalls, the code can be rearranged.



Write After Write (WAW) Hazard

Occurs when a later instruction tries to write an operand before an earlier instruction writes to it. The write ends up being performed in the wrong order, leaving the value written by the earlier instruction rather than the value written by the later instruction in the register. This type of hazard does not exist in the learned 5-stage in-order pipeline and can also be solved by register renaming.

Write After Read (WAR) Hazard

Occurs when a later instruction tries to write an operand before an earlier instruction. The earlier instruction incorrectly gets the new value written by the later instruction. This type of hazard occurs either when there are instructions that write results early in the instruction pipeline and other instructions read that source late in the pipeline, or when instructions are reordered. This type of hazard does not exist in the 5-stage in-order pipeline and can also be solved by register renaming.

Example: Pipeline Cycle Diagrams (No Forwarding)

Give the pipeline cycle diagram for the code segment below given a 5-stage pipeline **without** forwarding.

```
1. lw r2, 0(r1)
2. sw r3, 0(r2)
3. add r1, r2, r3
4. lw r5, 4(r1)
5. add r7, r5, r6
```

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14	C15
i1	IF	ID	EXE	MEM	WB										
i2		IF	**0**	**0**	ID	EXE	MEM	WB							
i3			**0**	**0**	IF	ID	EXE	MEM	WB						
i4			**0**	**0**		IF	**0**	**0**	ID	EXE	MEM	WB			
i5			**0**	**0**			**0**	**0**	IF	**0**	**0**	ID	EXE	MEM	WB

Explanation:

1. The first instruction i1 executes fine, there are no dependencies
2. The second instruction i2 depends on the register value r2 that is read from lw. Because there is no forwarding, we must wait all the way until WB to perform the ID stage. WB writes to registers in the first half of the clock cycle, and ID will read from them in the second half of the clock cycle. *The stalls propagate down (see C2 and C3) because while beginning IF for the next instruction would be possible, it could affect the order of completion for the instructions, which can cause long stalls.*
3. The third instruction i3 depends on r2 and r3. r3 is not written to in i1 or i3, so no concern. R2 is written to in i1, but that instruction is complete so no stalls.
4. The fourth instruction i4 depends on the result of r1, which is written in i3. Because there is no forwarding, we must wait until the WB stage of i3. Bubbles propagate down like above
5. The fifth instruction i5 depends on the result of r5, which is written in i4. So we must stall until WB stage of i4.

Forwarding

In instructions with data dependencies the, ALU will already contain the result needed for the next instruction, so as an optimization, we can pass the result of the ALU directly back, allowing us to overlap the EXE and ID instructions for data dependencies based on the ALU.

Note: dependencies based on memory access (like lw) must still wait for the MEM stage and will forward from the memory to the ALU

Example: Pipeline Cycle Diagrams (Forwarding)

Give the pipeline cycle diagram for the code segment below given a 5-stage pipeline **with** forwarding.

```
1. add r1, r2, r3
2. sub r2, r1, r2
3. and r2, r1, r4
4. slt r8, r2, r3
5. add r7, r5, r6
```

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
i1	IF	ID	EXE	MEM	WB									
i2		IF	ID	EXE	MEM	WB								
i3			IF	ID	EXE	MEM	WB							
i4				IF	ID	EXE	MEM	WB						
i5					IF	ID	EXE	MEM	WB					

[ ] = Forwarding

Explanation:

1. First instruction i1 executes fine, no dependencies
2. The second instruction r2 is dependent on r1 and r3. Instruction 1 writes to the r1 address, but we can directly forward the instruction in Clock Cycle 3. The EXE stage of i1 is completed in the first half of the clock cycle, and the result is directly forwarded back to the ALU for the ID stage of i2 in the second half of the clock cycle
3. The third instruction i3 depends on r1 and r4. No dependencies from the previous instruction, so no concern
4. The fourth instruction i4 depends on r2 and r3. Instruction i3 writes to r2, but we can forward the result from the EXE stage of i3 directly back to the ALU for the ID stage of i4
5. The fifth instruction i5 depends on r5 and r6. No dependencies, so no issue.

Give the pipeline cycle diagram for the code segment below given a 5-stage pipeline **with** forwarding.

```
1. lw r2, 0(r1)
2. sw r3, 0(r2)
3. add r1, r2, r3
4. lw r5, 4(r1)
5. add r7, r5, r6
```

	C1	C2	C3	C4	C5	C6	C7	C8	C9	C10	C11	C12	C13	C14
i1	IF	ID	EXE	MEM	WB									
i2		IF	**0**	ID	EXE	MEM	WB							
i3			**0**	IF	ID	EXE	MEM	WB						
i4			**0**		IF	ID	EXE	MEM	WB					
i5			**0**			IF	**0**	ID	EXE	MEM	WB			

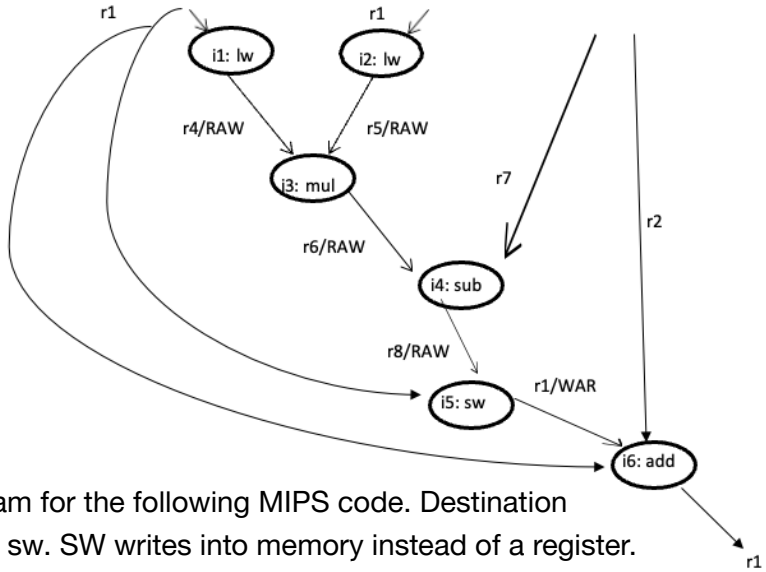
Explanation:

1. The first instruction i1, no dependencies, no issues
2. The second instruction i2 depends on r2 and r3. Because i1 reads a value from memory into r2, we must wait until the i1 MEM stage to decode i2. We can forward directly from the MEM stage in i1 to the ID stage in i2
3. The third instruction i3 depends on r2 and r3. We write to r2 in i1, but that instruction is long done, so no dependencies. The sw instruction in i2 does not change either r2 or r3 as it updates a memory address.
4. The fourth instruction i4 depends on r1. i3 writes to r1, but we can forward the ALU result back into itself, allowing us to execute the EXE stage of i3 in the first half of C5 and the ID stage of i4 in the second half of C5
5. The fifth instruction i5 depends on r5 and r6. Because in i4 we read a memory value into r5, we must wait until the MEM stage of i4 completes to begin the ID stage of i5. However, we can forward directly from the MEM stage to the ID stage in C8

Example: Draw a Data Dependency Diagram

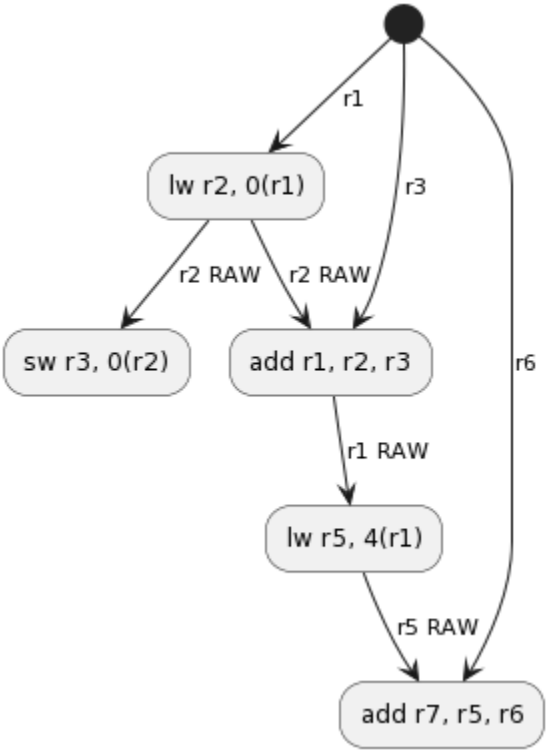
Given the following MIPS code, draw a data dependency diagram. Destination registers are listed first, except for the sw instructions (sw writes it into memory rather than a register).

```
i1: lw r4, 0( r1 )
i2: lw r5, 4( r1 )
i3: mul r6, r4, r5
i4: sub r8, r6, r7
i5: sw r8, 8(r1)
i6: add r1, r1, r2
```



Draw the data dependency diagram for the following MIPS code. Destination registers are listed first except for sw. SW writes into memory instead of a register.

```
lw r2, 0(r1)
sw r3, 0(r2)
add r1, r2, r3
lw r5, 4(r1)
add r7, r5, r6
```



`lw r2, 0(r1)`
Depends on r1, writes to r2

`sw r3, 0(r2)`
Depends on r2 and r3

`add r1, r2, r3`
Depends on r2 and r3, writes to r1

`lw r5, 4(r1)`
Depends on r1, writres to r5

`add r7, r5, r6`
Depends on r5 and r6, writes to r7

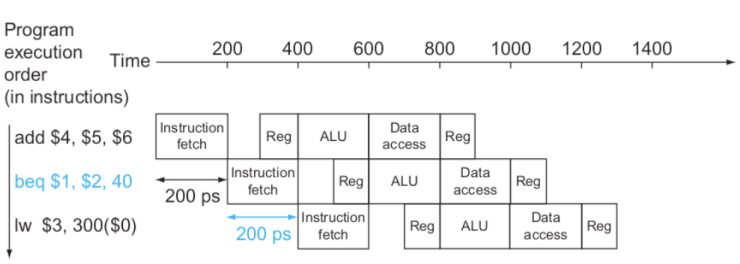
Branch Prediction

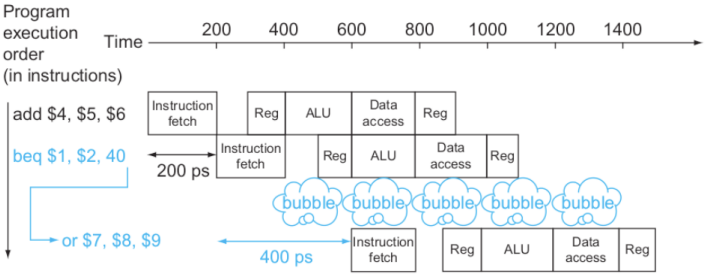
With branch prediction, we try to predict both the decision and the target address. It will speculatively execute instructions along the predicted path with one of two outcomes:

1. If the prediction is correct, there's no loss in time or data.
2. If the prediction is wrong, flush the bad instructions in the pipeline and fetch the instructions from the new branch.

Static Branch Prediction

Simplest scheme is to always predict "not taken" or always predict "taken" and continue executing down the instruction stream. Branches are often highly biased towards taken or untaken, and as such, profile information collected from earlier runs can be used to do compile-time branch prediction. This is called *profile-based prediction*.

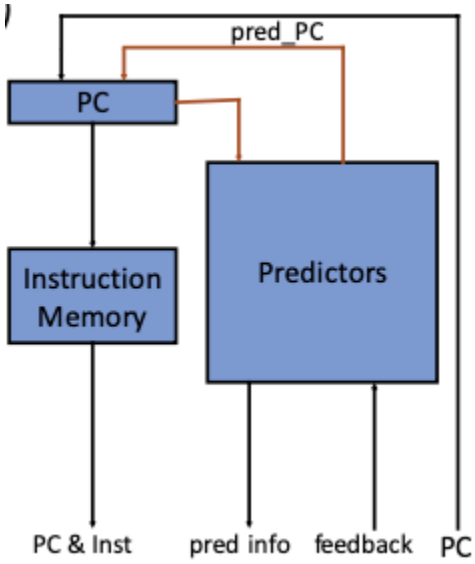




Dynamic Branch Prediction

This method of branch prediction can improve performance by using runtime information. By using the Program Counter and past branch history, we can predict:

- Conditional branch instruction's branch direction and target address
- Jump instruction's target address
- Procedure call/return's target address

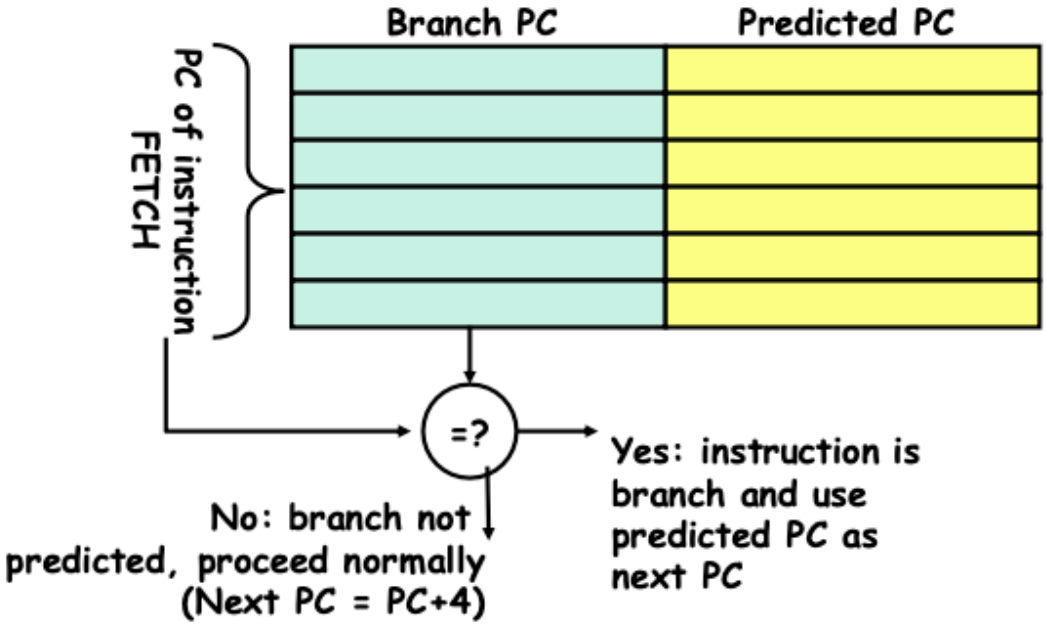


Out-of-Order Execution

Instructions are fetched and issued in compiler-generated order, however, instructions may be executed in some other order since independent instructions behind a stalled instruction can pass it.

Branch Target Buffer (BTB)

The branch target buffer caches the destination PC or destination instruction for a branch



Multiple Issue

Under ideal conditions, a single-issue CPU that implements pipelining and without hazards has an CPI (clocks per instruction) value of 1. A multiple-issue CPU can begin the execution of multiple instructions in the same clock cycle, allowing our Clocks Per Instruction value to theoretically drop under 1 (more than 1 instruction per cycle on average).

- ➔ **Static Multiple Issue**: Allowing the compiler to group instructions into issue groups
- ➔ **Dynamic Multiple Issue**: Implementing multiple issue dynamically using an instruction queue

Speculation

In order to more effectively utilize dynamic issue, we must apply techniques to increase the *number of instructions* without hazard to improve the efficiency. Speculation is a technique that utilizes *branch prediction* in order to make good use of the processors and executes instructions along the predicted path. Speculation is common to both static and dynamic multiple issue and can help if done well.

Loop Unrolling

Also called loop unwinding, loop unrolling is a technique that attempts to optimize a program's execution speed at the expense of its binary size. The optimizations done to make the code faster is reducing the loop-control overhead, which exposes the code to more parallelism. A technique called *register renaming* can be used as well, which uses different registers per replication (or per-loop).

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1, -16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	add $t0, $t0, $s2	lw $t2, 8($s1)	3
	add $t1, $t1, $s2	lw $t3, 4($s1)	4
	add $t2, $t2, $s2	sw $t0, 16($s1)	5
	add $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

What is the IPC?
IPC = 14/8 = 1.75 (close to 2)

Example: Matching

Words
Control Signal, Structural Hazard, Control Hazard, Speculative Execution, Out of Order Execution, Load-Use Data Hazard, Forwarding, IPC

1) Value for selecting a mux input or selecting the operation of a functional unity. **Control Signal**

2) When hardware cannot support the combination of instructions we want to execute in the same clock cycle. **Structural Hazard**

3) Allowing an instruction that is control dependent on a branch to execute after the branch direction is predicted and before the branch is resolved. **Speculative Execution**

4) Allowing instructions behind a stalled instruction to proceed to execution. **Out-of-Order Execution**

5) Data being loaded by a load instruction has not yet become available when it is needed by another instruction. **Load-Use Data Hazard**

6) Providing a data value to any unit where it is needed after the data value has been produced but before it is available in the register file. **Forwarding**

7) The measure of performance behind a stalled instruction to proceed to execution. **IPC**

Example: Short Answer

a) Does internal forwarding always eliminate all stall cycles due to hazards in a pipeline? You can justify/explain your answer by using a simple assembly or pseudocode

When memory access is needed, there are still some stall cycles when forwarding is used due to the load-use hazard.

```
1. lw r2, 0(r1)
2. sw r3, 0(r2)
3. add r1, r2, r3
```

b) What does the Branch Target Buffer (BTB) store, and how is this information used?

The BTB caches the destination PC or destination instruction for a branch. This allows the computer to predict the path the branch may take, and the destination PC can then be quickly obtained by the cache, improving branch performance.

c) Given a simple 5-stage MIPS pipeline with single issue, what is the ideal IPC and for which scenario?

*The ideal IPC for a single issue MIPS pipeline is **1**, but only when there are no dependencies*

d) What hazard does it resolve by using an instruction memory and a data memory in the datapath?

Structural Hazard

e) When MIPS pipeline uses extra hardware to compare two register values in the ID stage instead of using the ALU in the EXE stage, what hazard does it address and what does it improve?

When the MIPS pipeline uses extra hardware to compare two register values in the Instruction Decode stage, it addresses the data hazard known as the "RAW" (Read After Write) hazard.

In a MIPS pipeline, the RAW hazard occurs when an instruction in the Execute stage attempts to read a register that has not yet been written back by the previous instruction in the Write Back stage. This can lead to incorrect results since the register may not have the expected value yet.

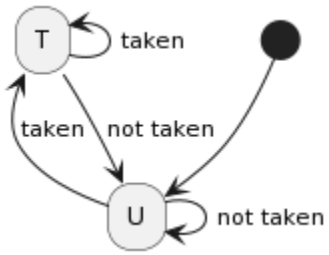
By comparing the register values in the Instruction Decode stage, the MIPS pipeline can detect when an instruction in the Execute stage is trying to read a register that has not

yet been written back. This allows the pipeline to stall the Execute stage until the necessary register value is available, preventing incorrect results and ensuring correct program behavior.

Using extra hardware for register comparison in the Instruction Decode stage can improve the performance of the MIPS pipeline by reducing the frequency of pipeline stalls. It can also improve the accuracy of program execution by avoiding incorrect results due to the RAW hazard.

f) Consider a one-bit history for branch prediction. It records the state of the last branch as taken (T) or untaken (U) and predicts the next branch will be the same. Assume the bit is unintialized to U. Determine the prediction accuracy on the following branch trace; include all trace entries in your calculation

We can use a sequential logic circuit to represent this system. Consider the Finite State Machine that represents this predictor.



Predicted	Actual	Correct	New Prediction
U	T	No	T
T	U	No	U
U	T	No	T
T	U	No	U
U	T	No	T
T	U	No	U
U	T	No	T
T	U	No	U

Now consider a 2-bit history predictor scheme