CS2100 AY23/24 SEM1 Midterms Cheat Sheet

How to detect Overflow:
- *If positive* add *positive* → *negative*
- *If negative* add *negative* → *positive*

char → 1 byte
int → 4 bytes
float/double → 4/8 bytes

C Programming

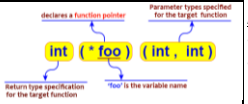
Format Specifiers: ** 1 Byte = 8 bits; 1 Nibble = 4 bits

Placeholder	Variable Type	Function Use
%c	char	printf / scanf
%d	int	printf / scanf
%f	float or double	printf
%f	float	scanf
%lf	double	scanf
%e	float or double	printf (for scientific notation)
%p	pointer	printf("&a = %p\n", &a)

Escape Sequence

Escape Sequence	Meaning	Result
\n	New line	Subsequent output will appear on the next line
\t	Horizontal tab	Move to the next tab position on the current line
\"	Double quote	Display a double quote "
%%	Percent	Display a percent character %

Symbols:
- ~ → NO; ^ → XOR; ! → boolean operator, not bitwise operator
- & -> address-of operator
- = → assign value on its right to the variable on its left(lvalue)



Arithmetic Operations: In C, % is NOT MODULO, its REMAINDER

Binary Operators: +, -, *, /, %	Unary Operators: +, -
Left Associative (from left to right)	Right Associative

Operator Type	Operator	Associativity
Primary Expression Operators	() *expr++ expr--*	Left to Right
Unary Operators	* & + - *++expr --expr (typecast)*	Right to Left
Binary Operators	* / %	Left to Right
	+ -	
Assignment Operators	= += -= *= /= %=	Right to Left

Syntax of Structure/ Array of Pointers

```
typedef struct {
    // members...
} structure_name;
```

- Use '.' to access the members of a structure.
- Use '->' to for pointers to structures
- Use '*' to declare a pointer to a structure

```
type
*array_name[size];
```

- ✅ (*player_ptr).name
- ✅ player_ptr->name
- ❌ *player_ptr.name // treated as *(player_ptr.name)

Number Systems
- N bits can represent up to 2^N values
- To represent M values, $\lceil \log_2 M \rceil$ bits required
- Decimal to Base-R
  - Whole numbers: repeated division-by-R
  - Fractions: repeated multiplication-by-R

Note that each upper-case alphabetical letter in ASCII differs by a value of 32 with its lower-case alphabetical letter.

Negative Numbers/ Negation of Numbers ***WORKS with FRACTIONS TOO!!

1s
- Technique to negate a value: invert all the bits. $-x = 2^n - x - 1$
- Range (for *n* bits): $-(2^{n-1} - 1)$ to $2^{n-1} - 1$
- The MSB represents sign: 0 → +'ve, 1 → -'ve

Weight of sign bit = $+/- 2^{n-1} - 1$

Addition (A+B)	Subtraction A-B = A + (-B)
• Add A + B normally in binary • If theres a carry out of MSB, + 1 • Check for overflow	• Take 1s-comp of B • Add 1s-comp B to A

2s
- Technique to negate a value: invert all the bits, then add 1. $-x = 2^n - x$
- Range (for *n* bits): -2^{n-1} to $2^{n-1} - 1$
- The MSB represents sign: 0 → +'ve, 1 → -'ve

Weight of sign bit = $+/- 2^{n-1}$

Addition (A+B)	Subtraction A-B = A + (-B)
• Add A + B normally in binary • Discard MSB overflow • Check for overflow	• Take 2s-comp of B • Add 2s-comp B to A

Excess Representation
- It allows the range of values to be distributed evenly between the positive & negative values.
- Divide the N-bits into 2 components. Top half represents –'ve numbers, lower half +'ve numbers
- Binary rep. of x in excess-N = Binary rep. of (x + N) ** i.e Excess-n → 000…0 = -n
- Dec rep. of x represented in excess-N: Convert x back to decimal and minus the result by N

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}
128	64	32	16	8	4	2	1	0.5	0.25	0.125	0.0625
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HEXA								A	B	C	D	E	F

Swap Function	XOR Swapping

`void swap(int *ptr1, int *ptr2) {`
`  int temp;`
`  temp = *ptr1;`
`  *ptr1 = *ptr2;`
`  *ptr2 = temp;  }`
	`void swap(int *a, int *b) {`
	`  *a = *a ^ *b;`
	`  *b = *a ^ *b;`
	`  *a = *a ^ *b;`
	`} * wont work if val is 0`

Floating Point Representation (IEEE 754- Single Precision Floating-Point Representation)

sign	exponent	mantissa

Sign (1bit)	Exponent (8bit)	Mantissa (23 bit)
• 1 bit • 0 → +'ve, 1 → -'ve - Sign is only for Mantissa	• 8 bit exponent (excess -127) • +'ve/-'ve power of the radix • E.g A x 10^4 = 127 + 4 = 131_{10}	• 23 bit • Significant digits of the floating-point number

- Convert to binary, normalize it (i.e standard form), and fill in the sign, exponent, mantissa
- Convert to hexa (depends on qns)

Pointers & Functions
- Addresses are printed out in hexadecimal format in C.
- A variable that contains the address of another variable is called a pointer variable/pointer.
- type *pointer_name; * → indicates a pointer
- Pointer Increment: (p = p + 1 (or p++))
  - If p is a pointer to an integer variable, then p = p + 1 (or p++) will increment the pointer by 4 bytes (assuming an integer is 4 bytes long) ** Depends on type, C auto * type size

Array, Strings & Structures
- An array name is a fixed pointer; it points to the first element of the array, and this cannot be altered.
- String: An array of characters, terminated by a null character '\0' (has an ASCII value of zero)
  - a string that is not properly terminated with '\0' will result in illegal access of memory.

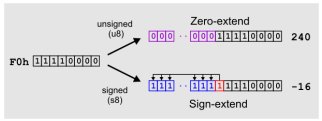
Read string from stdin (keyboard)	`fgets(str, size, stdin) // reads size - 1 char, // or until newline` `scanf("%s", str);  // reads until white space`

Print string to stdout (monitor)	`puts(str);  // terminates with newline`
	`printf("%s\n", str);`

fgets()	• On interactive input, fgets() also reads in the newline character
	• Hence, we may need to replace it with '\0' if necessary
	`fgets(str, size, stdin);`
	`len = strlen(str);`
	`if (str[len - 1] == '\n')`
	`  str[len - 1] = '\0';`

- Structures allow grouping of heterogeneous members (of different types)
  - A type is NOT a variable!
    - A type needs to be defined before we can declare variable of that type
    - No memory is allocated to a type

Sign Extension



MIPS
- Software executes sequentially BY DEFAULT, Hardware executes parallel BY DEFAULT
- Load-Store Model: Limit memory operations and relies on registers for storage during execution
  - i.e Only load & store access memory, the others operate on registers
- A typical assembly code structure is: (1) load, (2) compute, (3) store.

MIPS Registers ***REGISTERS DO NOT HAVE TYPES
- $at (register **1**) is reserved for the **assembler.**
- $k0-$k1 (registers **26-27**) are reserved for the **operating system**.

Logical operations/ Truth Table

Logical operation	C operator	Java operator	MIPS instruction
Shift Left	<<	<<	sll
Shift right	>>**	>>, >>>	srl
Bitwise AND	&	&	and, andi
Bitwise OR	|	|	or, ori
Bitwise NOT*	~	~	nor
Bitwise XOR	^	^	xor

MIPS CheckList:

Operation	Opcode in MIPS	Meaning
Addition	add $rd, $rs, $rt	$rd = $rs + $rt
	addi $rt, $rs, $C16_2s$	$rt = $rs + $C16_2s$
Subtraction	sub $rd, $rs, $rt	$rd = $rs - $rt
Shift left logical	sll $rd, $rt, C5	$rd = $rt << C5
Shift right logical	srl $rd, $rt, C5	$rd = $rt >> C5
AND bitwise	and $rd, $rs, $rt	$rd = $rs & $rt
	andi $rt, $rs, C16	$rt = $rs & C16
OR bitwise	or $rd, $rs, $rt	$rd = $rs | $rt
	ori $rt, $rs, C16	$rt = $rs | C16
NOR bitwise	nor $rd, $rs, $rt	$rd = $rs ↓ $rt
XOR bitwise	xor $rd, $rs, $rt	$rd = $rs ^ $rt
	xori $rt, $rs, C16	$rt = $rs ^ C16

C5 is [0 to 2^5-1] $C16_{2s}$ is [-2^{15} to 2^{15}-1] C16 is a 16-bit pattern

MIPS are all in 2s complement!!

AND

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

NOTE: 1 if BOTH a and b are 1. Otherwise 0.

NOR

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

NOTE: 1 if BOTH a and b are 1. Otherwise 0.

OR

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

NOTE: 0 if BOTH a and b are 0. Otherwise 1.

XOR

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

NOTE: 1 if a is NOT the same as b.

- sll and srl only need 5 bits (i.e., C5) because shifting by 32-bits empties the register (i.e., set to 0).
- C16 are NOT sign-extended.
- $C16_{2s}$ are sign-extended.

ONLY binary has the behaviour where the MSB represents -2^{n-1}
Diminished Radix: **r's complement = r^n – number**
(r-1)'s complement = r^n – 1 – number

Word Alignment (Check if Address of Word % word size = 0)
- a single **byte** (**byte addressable**) or ** char = 1 byte
- a single **word** (**word addressable**) **1 word = 4 bytes (Word is usually 2^n bytes)
  - word alignment → the memory address is a multiple of the word size
- NOTE: MIP uses byte addressing (conseq words differ by 4 bytes, Mem[0], Mem[4])

Branching (Uses PC-relative addressing)
- If condition = true → redirect to the branch labelled "X" (something like a checkpoint)
  - Note: A label is NOT an instruction
- `immediate` field specifies the number of words to jump (i.e No. of instructions to "skip over")
- `immediate` field can be positive or negative
- If branch is not taken: PCnew = PC + 4
- If branch is taken: PCnew = (PC + 4) + (immediate x 4)

Jump (j L1) → Jump to label "L1" unconditionally **equivalent to beq $s0, $s0, L1 (uses pseudo addrs)
- Processor always follows the branch
- Given address of instruction, drop the first 4 & last 2 bits, the rest will be your immediate field.

R-Format (6 + 5 + 5 + 5 + 5 + 6 = 32 bits) **shamt → shift amount 😊

Field Bits	6	5	5	5	5	6
Field Name	opcode	rs	rt	rd	shamt	funct

- Each field is an independent 5- or 6- unsigned integer
  - A 5-bit field can represent any number 0 – 31
  - A 6-bit field can represent any number 0 – 63

Arithmetic	Shift
`arith $rd, $rs, $rt` • opcode is ALWAYS 0 • shamt is ALWAYS 0 • arith is arithmetic operation opcode | rs | rt | rd | shamt | funct 0 | rs | rt | rd | 0 | XX	`shift $rd, $rt, shamt` • opcode is ALWAYS 0 • rs is ALWAYS 0 • shift is shift operation opcode | rs | rt | rd | shamt | funct 0 | 0 | rt | rd | shamt | XX

I-Format (6 + 5 + 5 + 16 = 32 bits)

Field Bits	6	5	5	16
Field Name	opcode	rs	rt	immediate

- Only 1 field is different from R-format → immediate
- The 3 others, `opcode`, `rs` and `rt` are still in the same locations (`rd`, `shamt` & `funct` merged)

Arithmetic	Load/Store
`arith $rt, $rs, C16_2s` • $C16_{2s}$ is in 2s complement • arith is arithmetic operation opcode | rs | rt | immediate XX | rs | rt | $C16_{2s}$	`ld/st $rt, $rs, C16_2s` • $C16_{2s}$ is in 2s complement • ld/st is a load or store operation opcode | rs | rt | immediate XX | rs | rt | $C16_{2s}$
Logic	**Branch**

`logic $rt, $rs, C16`
• C16 is raw binary (*no negative*)
• logic is logical operation (*bitwise operation*)
opcode | rs | rt | immediate
XX | rs | rt | C16
	`branch $rs, $rt, label`
	• branch is a branch operation
	• label is converted to number first (*PC-relative addressing*)
	  - Note the position of rs and rt here.
	  - First register is NOT rt but is rs instead!
	opcode | rs | rt | immediate
	XX | rs | rt | label

J-Format (6 + 26 = 32 bits) ** FOR JUMP INSTRUCTIONS

Field Bits	6	26
Field Name	opcode	target address

- Keep `opcode` field identical to R-format & I-format for consistency
- Combine all other fields to make room for a larger target address
  - Specify only 26 bits of 32 bits (Need to drop the first 4 bits and last 2 bits if convert to binary)

Note:
- Jumps will only skip to word-aligned addresses (x4), so last 2 bits are always 00
- Hence, we just assume the address ends with 00, and we can specify 28 bits of 32 bits
- The 26bits is left shifted by 2, giving it a relative address boundary of 2^{28} (Max range: 2^{28} = 256MB) or (Max range: 2^{26} instructions)

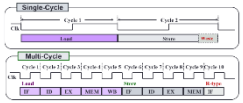
PC ↔ J-Format Converter

PC → J-Format		J-Format → PC
Step 1:	opcode | target ad.	Do the reverse
Step 2:	Delete 1st 4, last 2 bits	• Append the 1st 4 MSB (ABCD) from PC + 00 for last 2 bits to the target address.
Step 3:	Combine to J-format	

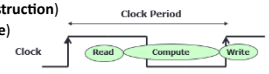
PC: ABCD <target_address_26_bits>00
J-Format: <opcode_6_bits>< ~~ABCD~~< target_address_26_bits >~~00~~>

The J instruction allows you to jump within a range of 2^{26} instr, but you can still jump to instr with addr starting with 1 or within the 256MB address boundary, if word alignment and the PC+4 relationship is accounts for. (j -1 == j 0x10000000)

- **RegDst**: 0 -> choose rt; 1-> choose rd
- **ALUSrc**: 0 -> choose RD2; 1-> choose SignExt(Inst[15:0])
- **MemToReg:** 0 -> choose ALU result; 1-> choose Memory Read Data
- **PCSrc**: 0 -> PC + 4; 1 -> (PC+4) + SignExt(Inst[15:0]) << 2

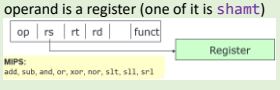


Uniform processing time → Use single cycle. (clock period = length of slowest **instruction**)
Variable processing time → Use Multicycle (clock period = length of slowest **stage**)

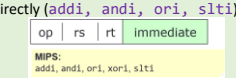


Addressing Modes *** MIPS only has 3 main addressing mode

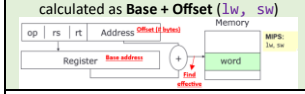
Register
- operand is a register (one of it is `shamt`)



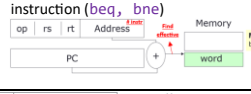
Immediate
- operand is specified in the instructions directly (`addi, andi, ori, slti`)



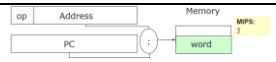
Displacement (Base)
- operand is in memory with address calculated as **Base + Offset** (`lw, sw`)



PC-relative
- address is sum of PC and constant in the instruction (`beq, bne`)



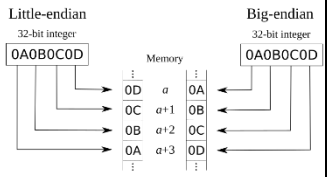
Pseudo-direct
- 26-bit of instruction concatenated with upper 4-bits of PC (`j`)



- Branches and load/store are both I-format instructions; but branches use PC-relative addressing, whereas load/store use base addressing
- Branches use ***PC-relative*** addressing
- Jumps use ***pseudo-direct*** addressing

Note:
- `beq, bne`: ignore PC, count displacement
- `j`: use PC, ignore displacement

Addressing mode	*Example*	*Meaning*
Register	Add R4,R3	R4 ← R4+R3
Immediate	Add R4,#3	R4 ← R4+3
Displacement	Add R4,100(R1)	R4 ← R4+Mem[100+R1]
Register indirect	Add R4,(R1)	R4 ← R4+Mem[R1]
Indexed / Base	Add R3,(R1+R2)	R3 ← R3+Mem[R1+R2]
Direct or absolute	Add R1,(1001)	R1 ← R1+Mem[1001]
Memory indirect	Add R1,@(R3)	R1 ← R1+Mem[Mem[R3]]
Auto-increment	Add R1,(R2)+	R1 ← R1+Mem[R2]; R2 ← R2+d
Auto-decrement	Add R1,–(R2)	R2 ← R2-d; R1 ← R1+Mem[R2]
Scaled	Add R1,100(R2)[R3]	R1 ← R1+Mem[100+R2+R3*d]



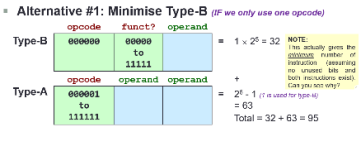
ISA (Instruction Set Architecture)

von Neumann	Data (operands) are stored in memory
Stack	Operands are implicitly on top of the stack
Accumulator	One operand is implicitly in the accumulator (a special register).
General Purpose Register	• Only explicit operands. • Register-memory architecture (one operand in memory) • Register-register (or load-store) architecture (MIPS)
Memory-memory	All operands in memory
Big-Endian	MSB stored in lowest address (MIPS)
Little Endian	LSB stored in lowest address (easier to read)

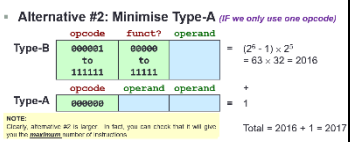
OP Code (Maximising/Minimising) ****Rich get richer**, to maximise, max out the one with more bits

Formula for Max: $2^{\#no.bits} \times (1 - F)$
- F → fraction of bits lost by reserving bits

Alternative #1: Minimise Type-B *(IF we only use one opcode)*

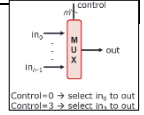


Alternative #2: Minimise Type-A *(IF we only use one opcode)*



Data Paths

Fetch	• Instruction Fetch Stage: - In this stage, the processor fetches the instruction from memory (usually the instruction cache) using the program counter (PC) to determine the address of the next instruction. - The fetched instruction is then passed to the next stage.
Decode	• Instruction Decode Stage: - During this stage, the fetched instruction is decoded to determine the operation to be performed, the source operands, and the destination register. - This stage may also involve reading values from the register file.
ALU	• Arithmetic Logic Unit stage (aka Execution Stage) - In this stage, arithmetic and logic operations are performed. - The ALU (Arithmetic Logic Unit) computes the result of the operation, which may include addition, subtraction, logical operations, etc. - The result is typically stored in a temporary register.
Memory	• Memory Access Stage: - If the instruction requires memory access (e.g., load or store operations), this stage is responsible for reading from or writing to memory. - It also handles address calculations and data transfers between the processor and memory.
RegWrite	• Instruction Register Write Back Stage - In the final stage, the result of the operation is written back to the destination register specified by the instruction. - This stage also handles updating the PC and preparing for the next instruction to be fetched.



Multiplexer:

A another combinational logic, to select which input is passed to the output.

Function	Selects one input from multiple input lines
Inputs	**n** lines of same width
Outputs	**m** bits where n = 2^m
Control	• One control with a width of $\log_2$**n** shown by the control line at the top/bottom • This controls which input is selected to output (Select i^{th} input line if control = i)

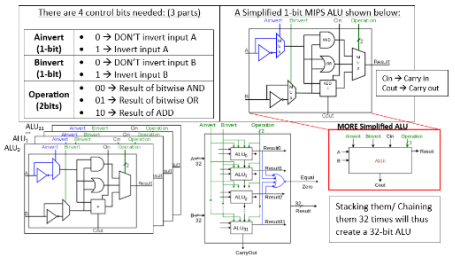
ALU Branch Instruction

1. **Branch Outcome:**
   - Use ALU to compare the register
   - The 1-bit "`isZero?`" signal is enough to handle equal/not equal check (how?)
2. **Branch Target Address:**
   - Introduce additional logic to calculate the address
   - Need **PC** (from **Fetch Stage**)
   - Need **Offset** (from **Decode Stage**)

PCSrc Control
- 1 → PC = (PC + 4) + (immediate + 4)
- 0 → PC = (PC + 4)

The Processor Control

One Bit at a Time ** The instruction will then be fed to ALU



Instruction	Code
Addition	`add $rd, $rs, $rt`
Subtraction	`sub $rd, $rs, $rt`
Bitwise AND	`and $rd, $rs, $rt`
Bitwise OR	`or $rd, $rs, $rt`
Set on Less Than	`slt $rd, $rs, $rt`
Branch on Equal	`beq $rd, $rs, label`
Load Word	`lw $rt, offset($rs)`
Store Word	`sw $rt, offset($rs)`

ALUcontrol			
Ainvert	Binvert	Operation	Function
0	0	00	AND
0	0	01	OR
0	0	10	add
0	1	10	subtract
0	1	11	slt
1	1	00	NOR

Generating ALUControl Signal:

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	xxxxxx	add	0010
sw	00	store word	xxxxxx	add	0010
beq	01	branch equal	xxxxxx	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

- IF ALUop is 00 or 01 → ALUControl will not bother with the 6 bits R-format

Control Design: Inputs

	Opcode (Op[5:0] == Inst[31:26])						
	Op5	Op4	Op3	Op2	Op1	Op0	Value in Hexadecimal
R-type	0	0	0	0	0	0	0
lw	1	0	0	0	1	1	23
sw	1	0	1	0	1	1	2B
beq	0	0	0	1	0	0	4

	opcode	Ctrl
R-Format	000000	100100010
lw	100011	011110000
sw	101011	X1X001000
beq	000100	X0X000101

Control Design: Outputs

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

- Instruction Execution =
  1. Read contents of one or more storage elements (register/memory)
  2. Perform computation through some combinational logic
  3. Write results to one or more storage elements (register/memory)
- All these performed **within a clock period**

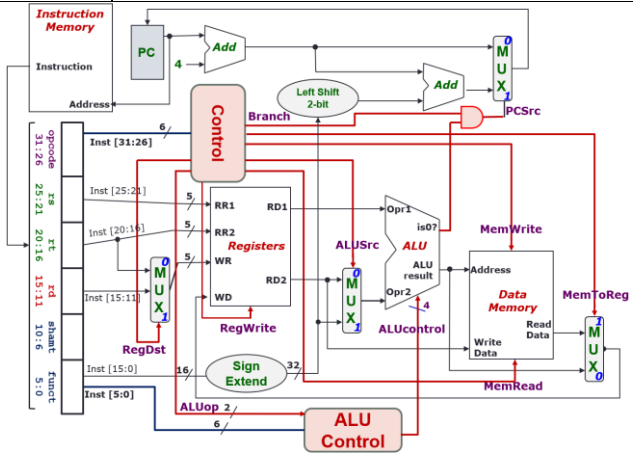
- A and B are inputs from registers/immediates.
- To correctly process these values (A and B) in the ALU, the ALUcontrol is used.
- ALUControl transcribes the ALUop code + funct into machine readable patterns of 1 & 0. It provides the instructions for the ALU on how to deal with the inpute A and B
- Example: ALUControl provides 0010. Then the ALU will know that we need to add A and B together.

Note: cycle time → match the slowest time
- Calculate cycle time assumes negligible delays ** Add the time taken tgt to get the time for instruction
- All instructions take as much time as the SLOWEST instruction (e.g load) ** Path taken

R-type	Inst-mem → Reg-file → MUX(ALUSrc) → ALU → MUX(MtoR) → Reg-file (e.g SUB) ↳ control (Not critical path)
LW/SW	Inst-mem → Reg-file → ALU → DataMem → MUX(MtoR) → Reg-file * Can ignore sign extend & ALUSrc since it is faster than RegFile (Usually, see the qns. If > RegFile, replace RegFile with signextend + ALUSrc)
BEQ	Inst-mem → Reg-file → MUX(ALUSrc) → ALU → AND → MUX(PCSrc)

- Solutions: Multicycle Implementation & Pipelining

Solution 1: Multicycle Implementation	• Break up the instructions into **execution steps**: 1. Instruction fetch 2. Instruction decode and register read 3. ALU operation 4. Memory read/write 5. Register write • Each execution step **takes one clock cycle** ➔ **Cycle time is much shorter, i.e., clock frequency is much higher** • Instructions take variable number of clock cycles to complete execution
Solution 2: Pipelining	• Break up the instructions into execution steps one per clock cycle • Allow different instructions to be in different execution steps simultaneously



0x0285c822 = **sub $25, $20, $5**; next PC = PC+4

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	WriteData
$20	$5	$25	[$20]-[$5]	[$20]	[$5]	[$20]-[$5]	[$5]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
1	1	0	0	0	0	0	10	0110

0x8df8000 = **lw $24, 0($15)**, next PC = PC+4

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	WriteData
$15	$24	$24	MEM([$15] + 0)	[$15]	0	[$15]+0	[$24]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
0	1	1	1	0	1	0	00	0010

0x1023000C = **beq $1, $3, 12**; next PC = PC+4 or (PC+4)+(12×4)

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	WriteData
$1	$3	$3 /$0	Random val	[$1]	[$3]	[$1]-[$3]	[$3]

RegDst	RegWr	ALUSrc	MRd	MWr	MToR	Brch	ALUop	ALUctrl
X	0	0	0	0	X	1	01	0110

Common Pseudocodes:

bgt $r1, $r2, L slt $t1, $r2, $r1 bne $t1, $zero, L	**bge $r1, $r2, L** slt $t1, $r1, $r2 beq $t1, $zero, L	**li $r, imm** (where imm can be 32 bits) • if less than/eq 16 bits ori $r $zero imm • if more then 16 bits: lui $r imm_U$ ori $r $r imm$_L$
ble $r1, $r2, L slt $t1, $r2, $r1 beq $t1, $zero, L	**nop** (null opr) - Do nothing sll $zero $zero 0	

Instructions Length: ** With C → 3 Types

$$Min = 2^{B-A} - 1 + 2^A \quad OR \quad 2^{C-B} - 1 + 2^{B-A} - 1 + 2^A$$
$$Max = 2^B - 2^{B-A} + 1 = (2^A - 1)(2^{B-A}) + 1 \quad OR \quad 2^C - 2^{C-B} + 1 - 2^{C-A} + 1$$