Midterm Cheat Sheet

By Phanuphat Srisukhawasu
CS2100 Taken in AY2024/25 S2
Last updated: March 11, 2025

Number Systems

Compatible Base Checking

Case 1: Checking base x for addition/subtraction operations.

1. Consider the first digit from LSB where adding digits cause overflow, i.e. the sum is lesser than one of the original values.
2. Apply the formula: `(digit1 + digit2) % x = resultDigit` and find x from here.

Case 2: Checking base for other operations. These problems mostly require brute force.

Complement Systems

Given that n/m is the number of integer/fractional digits

- **(r-1)'s complement**: $X' = r^n - r^{-m} - X$ (ranging from $-(r^{n-1} - 1)$ to $r^{n-1} - 1$).
- **r's complement**: $X'' = r^n - r^{-m} - X + 1$ (ranging from $-r^{n-1}$ to $r^{n-1} - 1$).

> **☰ Example**
>
> Calculate $0101.11 - 010.0101$ (in binary).
>
> **Solution**:
>
> - $0101.1100 - 0010.0101 = 0101.1100 + (2^4 - 2^{-4} - 0010.0101)$
> - $= 0101.1100 + (10000 - 0.0001 - 0010.0101) = 0101.1100 + (1111.1111 - 0010.0101)$
> - $= 0101.1100 + 1101.1010 = (1)0011.0110 = 0011.0111$
>
> Calculate 10's complement of -1 in 4 bits representation.
>
> **Solution**: -1 can be represented in 9's complement as $10^4 - 10^{-0} - 1 = 9998$ and as $9998 + 1 = 9999$ in 10's complement.

> **⚠ Warning**
>
> 1. In base-r complements, digit weights are not applicable except when $r = 2$.

2. Always extend the digits in both integer and fractional parts to match the other operands, as shown in the first example.
3. (r–1)'s complement propagates a carry-out to the end, whereas r's complement will just ignore the carry.

IEEE-754 Representation

☰ Example

Find the decimal value of `0xC4007000`.

Solution:

1. Use the calculator to convert from hex to binary (group by 1, 8, 23 bits): `0b1 10001000 00000000111000000000000`.
2. Read the first bit (`0` is positive and `1` is negative).
3. The next 8 bits are the exponent in Excess-127 format. `0b10001000` = 136 (Excess-127) = 136 - 127 = 9 (in decimals).
4. Write the expression as $\pm 1.XX \ldots X \times 2^n$ where X's are the remaining 23 bits = $-1.00000000111 \times 2^9$ which is `-0b1000000001.11` (like scientific notation).
5. Convert the resulting binary bits into decimal using the calculator: 513.75.

Note: Decimal + 127 = Excess-127 and Excess-127 - 127 = Decimal. This formula can be useful when converting the number back (from step 5. to 1.)

�™ Tip

- The **range** of the Excess-M system is from $-(M-1)$ to M.
- The smallest positive number representable in the IEEE-754 format is given by: $1.00 \ldots 0 \times 2^{-126}$.
- The most negative number representable in the IEEE-754 format is given by: $-1.11 \ldots 1 \times 2^{127}$.

C Programming

C always uses pass-by-value, but we can simulate **pass-by-reference with pointers**.

⚠ Warning

1. An array name (`arr`) is a fixed pointer to its first element (`&arr[0]`), meaning you can't reassign it (`arr1 = arr2` is invalid).

MIPS Programming

Instruction Encoding & Decoding

- **Encoding**: Refer to the instruction sheet, write the entire instruction in binary, and then convert to hex using the calculator.
- **Decoding**: Convert hex to binary using the calculator. Write the encoded instruction. After noting the first 6 bits, read the actual opcode to determine the type of the instruction. The subsequent groupings depend on whether it is R (5/5/5/5/6), I (5/5/16), or J (26) format.

Register

Aside from the constant zeroes (`$zero`), we have `t` for temporaries and `s` for saved temporaries.

$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$t8	$t9
01000	01001	01010	01011	01100	01101	01110	01111	11000	11001
$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7		
10000	10001	10010	10011	10100	10101	10110	10111		

R-Format

The instruction can be determined using the `funct` field. The shift amount (`shamt`) is only applicable to shift left/right logical instructions.

Mnemonic	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
`add rd, rs, rt`	000000	rs	rt	rd	00000	100000
`sub rd, rs, rt`	000000	rs	rt	rd	00000	100010
`sll rd, rt, shamt`	000000	00000	rt	rd	shamt	000000
`srl rd, rt, shamt`	000000	00000	rt	rd	shamt	000010

Mnemonic	opcode (6)	rs (5)	rt (5)	rd (5)	shamt (5)	funct (6)
and rd, rs, rt	000000	rs	rt	rd	00000	100100
or rd, rs, rt	000000	rs	rt	rd	00000	100101
xor rd, rs, rt	000000	rs	rt	rd	00000	100110
nor rd, rs, rt	000000	rs	rt	rd	00000	100111
slt rd, rs, rt	000000	rs	rt	rd	00000	101010

> 👌 **Tip**
>
> The opcode of the R format instruction is always `000000`.

I-Format

The immediate is always a 16-bit integer. You need to use load upper immediate (`lui`) with `ori` (for the lower 16 bits) to extend it to 32 bits.

Mnemonic	opcode (6)	rs (5)	rt (5)	immediate (16)
beq rs, rt, relative address	000100	rs	rt	number of words
bne rs, rt, relative address	000101	rs	rt	number of words
addi rt, rs, immediate	001000	rs	rt	immediate
andi rt, rs, immediate	001100	rs	rt	immediate
ori rt, rs, immediate	001101	rs	rt	immediate
xori rt, rs, immediate	001110	rs	rt	immediate
lui rt, immediate	001111	00000	rt	immediate
lb rt, immediate(rs)	100000	rs	rt	immediate
lw rt, immediate(rs)	100011	rs	rt	immediate
sb rt, immediate(rs)	101000	rs	rt	immediate
sw rt, immediate(rs)	101011	rs	rt	immediate

> ⚠️ **Warning**
>
> The number of words in the branch instruction is measured relative to `PC + 4`. That is, we jump to `(PC + 4) + (Immediate * 4)` if the branch is taken.

J-Format

The memory address is always 32 bits. However, since it must be well-aligned with offsets as multiples of 4, the last 2 bits can be ignored.

Mnemonic	opcode (6)	address (26)
`j address`	000010	26-bit target address (shifted left by 2 when used)

> ⚠️ **Warning**
>
> 1. The full address is formed using the upper 4 bits of `PC + 4`. This can cause jump instructions to fail if `PC` is near a boundary—specifically, when the upper 4 bits of `PC + 4` differ from those of `PC`.
> 2. The maximum jump range in bytes is 2^{28} from `PC + 4`. In general, it follows the formula: $2^{\text{immediate}} \times \text{word size}$ (4 in MIPS).

Instruction Set Architecture (ISA)

Maximum and Minimum Number of Instructions

Case 1: There is at least 1 ins truction on each instruction type.

> ≔ **Example**
>
> There are three types of instructions: **A (4-bit opcode), B (7-bit opcode), and C (8-bit opcode)**. Find the maximum and minimum total number of instructions.
>
> **Solution**:
>
> 1. Max $= (2^8 - 2^{8-7} - 2^{8-4}) + (1) + (1) = 240$ (Maximize C / Minimize A and B)
> 2. Min $= (2^4 - 1) + (2^{7-4} - 1) + (2^{8-7}) = 24$ (Maximize A and B / Minimize C)

Case 2: Each instruction type has a minimum required number of instructions, which may vary, but some must be greater than 1.

> ≔ **Example**
>
> There are three types of instructions: **X (2-bit opcode), Y (4-bit opcode), and Z (7-bit opcode)**. We need at least 2 X-Type and Y-Type Instruction with at least 1 Z-Type Instruction. Find the maximum and minimum total number of instructions.
>
> **Solution**: We use a similar approach as above but scale the relevant terms based on the number of instructions allocated for each instruction type.
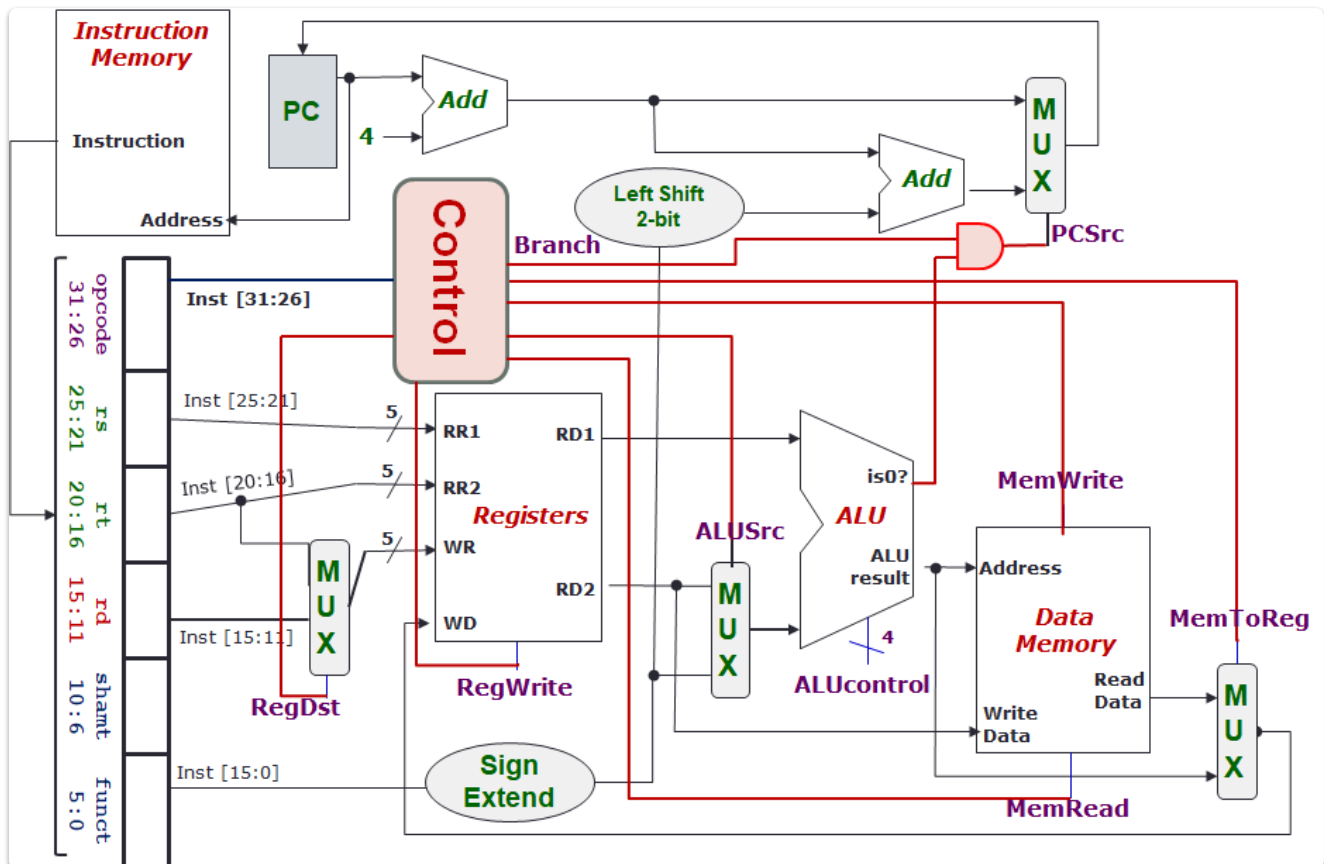
1. Max = $(2^7 - 2^{7-4}(2) - 2^{7-2}(2)) + (2) + (2) = 48 + 2 + 2 = 50$.
2. Min = $(2 + (2^2/2 - 1)) + (2 + (2^{4-2}/2 - 1)) + (2^{7-4}) = 3 + 3 + 8 = 14$.

In this example, we need to allocate two instructions for X and Y. Note that the minimum number of instructions does not need to be a power of two.

MIPS Datapath and Control

General Path

Refer to this diagram to trace the instruction execution path.

> ⓘ **Info**
>
> The multiplexer `MemToReg` is reversed only because the wires cross on the diagram.

> 👌 **Tip**
>
> The standard control signals for different types of instruction are shown below.
>
> | Instruction | RegDst | ALUSrc | MemToReg | RegWrite |
> | --- | --- | --- | --- | --- |
> | R-type | 1 | 0 | 0 | 1 |
> | lw | 0 | 1 | 1 | 1 |

Instruction	RegDst	ALUSrc	MemToReg	RegWrite
sw	X	1	X	0
beq	X	0	X	0

Instruction	MemRead	MemWrite	Branch	ALUop1	ALUOp0
R-type	0	0	0	1	0
lw	1	0	0	0	0
sw	0	1	0	0	0
beq	0	0	1	0	1

Critical Path

☰ Example

Given below are the resource latencies of various hardware components in picoseconds (ps): **Inst-Mem** (400 ps), **Adder** (100 ps), **MUX** (30 ps), **ALU** (120 ps), **Reg-File** (200 ps), **Data-Mem** (350 ps), **Control/ALU Control** (100 ps), **Left-shift/Sign-Extend/AND** (20 ps). Determine the latency for the instruction `lw $24, 0($15)`.

Solution

- **Fetch stage**: Fetching the instruction from memory takes **400 ps**. In parallel, `PC + 4` is computed using an adder, costing 100 ps, but this is not critical.
- **Decode stage**:
    - Reading the opcode to determine the instruction type and field lengths takes no time.
    - Reading data from the register file takes **200 ps**.
    - The control unit determines control signals and propagates them in **100 ps**, but this is not critical.
    - MUX inputs are pre-determined, so the `RegDst` and `ALUSrc` MUX takes no additional time later. However, other MUXs still need to wait for the input.
- **ALU stage**: Computing the memory address using the ALU takes **120 ps**.
- **Memory stage**: Reading data from memory takes **350 ps**.
- **Register write stage**: The result passes through the `MemToReg` MUX and is written to the register file, taking **30 + 200 = 230 ps**.

**Total latency**: $400 + 200 + 120 + 350 + 230 = 1300$ ps.

⚠ **Warning**

For branch instructions, there is a parallel execution of steps 2 and 3, which involve the following:

1. Sign-extending the immediate (20 ps),
2. Left-shifting by 2 (20 ps),
3. Adding to `PC + 4` (100 ps).

This combined operation costs a total of 140 ps, which is less than the combined latency of steps 2 and 3 in the above example.

After execution, the branch instruction waits at the `PCSrc` MUX for the `is0?` signal from the ALU, which is ANDed with the branch signal (20 ps) before passing through the MUX (30 ps).

ALU Slice
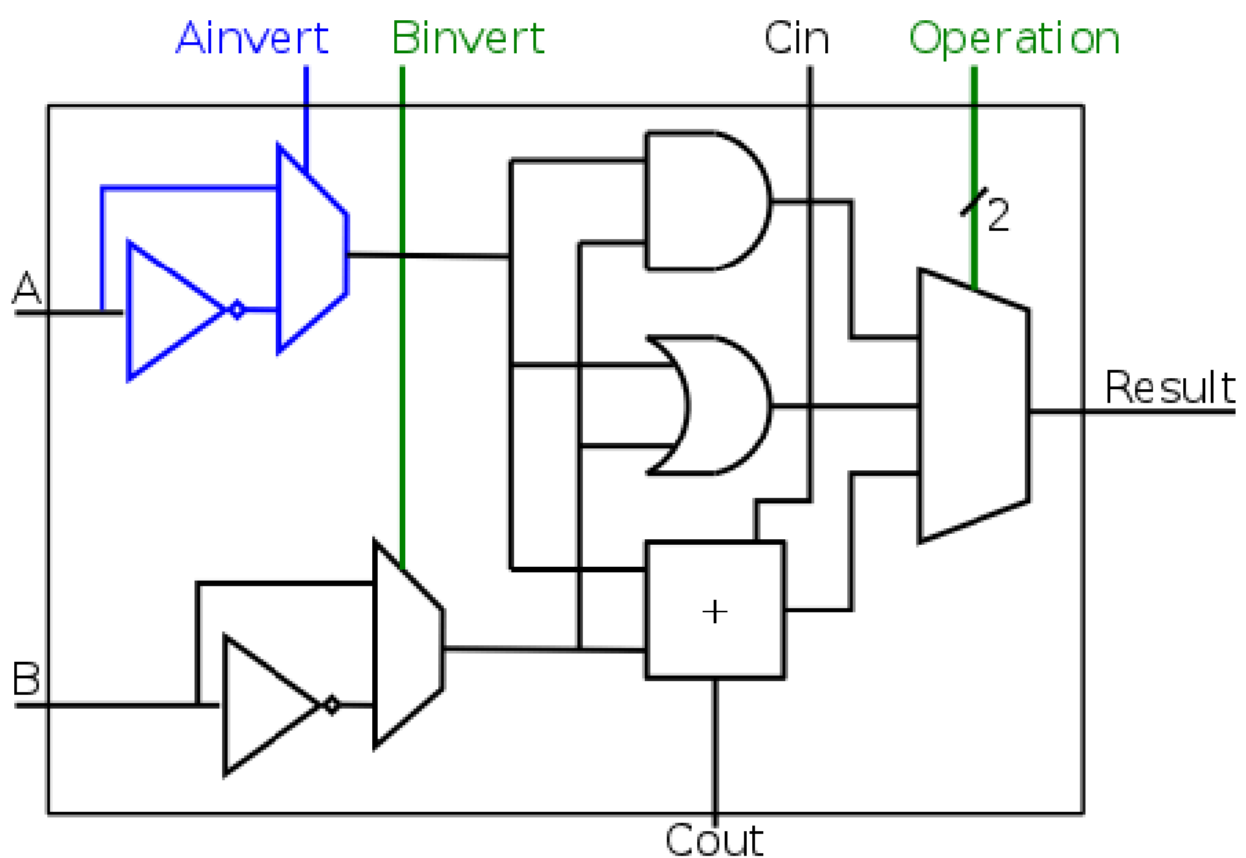
> ☝ **Tip**
>
> The **standard ALUControl signals** for different types of instruction are shown below.
>
> | Instruction | ALUControl |
> |---|---|
> | `lw` | `0010` |

Instruction	ALUControl
sw	0010
beq	0110
add	0010
sub	0110
and	0000
or	0001
slt	0111

Note:

- The first two bits indicate A inverse and B inverse. B inverse is `1` only for subtraction.
- The last two bits follow the ALU slice's operation order: `00` for `and`, `01` for `or`, `10` for `add`, and `11` for `slt` (hidden).

☰ Example

Given that all logic gates take 1 ps (picosecond) and MUXs take 2 ps, determine the maximum latency of a 4-bit ALU.

Solution:

- Inputs A and B arrive in parallel. The longest delay comes from inverting both, which takes $\max\{1+2, 1+2\} = 3$ ps.
- All operation gates also run in parallel, taking $\max\{1, 1, 1\} = 1$ ps for bit 0. for bit 0. However, carry propagation occurs from LSB to MSB. Since all slices operate in parallel, bits 1, 2, and 3 must wait 1 ps per previous bit. This delay accumulates, making the critical path for the MSB take $1 + 1 + 1 = 3$ ps.
- The total time so far is 6 ps. After passing through the operation MUX, the final delay is $6 + 2 = 8$ ps.

Good to Memorize

- **No Operation (NOP)** can be implemented by an instruction that avoids reading/writing to memory or modifying registers.
- **Register File** is a set of 32 registers, excluding immediate values.
- **Instruction Register (IR)** holds the encoded instruction currently being executed.
- **Special Register**: The stack pointer (`$sp`) points to the last occupied location at the top of the stack, which grows downward in memory.

- **Rising Edge of the Clock Cycle**: The moment when the program counter (`PC`) is updated.
- **Single-Cycle Implementation**: The cycle time is determined by the <mark>slowest instruction</mark>.
- **Multi-Cycle Implementation**: Each instruction is broken into steps, with each step taking one cycle. The overall cycle time depends on the <mark>slowest step</mark>.
- **Implementation of `slt`**: We get the <mark>sign bit from bit 31 and carry that to be bit</mark> as well as setting the remaining bits to be 0. If the result is negative, bit 0 will be 1.
- **Endianness** refers to the order in which bytes are arranged in a multi-byte word stored in memory. In **big-endian** format, the <mark>most significant byte (MSB) is stored at the lowest memory address</mark>, while in **little-endian** format, the <mark>least significant byte (LSB) is stored at the lowest memory address</mark>.