Chapter 4 – Processor

1. Rewind of previous notes

> **CPU Performance factors** is defined by 3 factors: instruction count *(based on ISA and compiler)*, CPI and Cycle time *(based on CPU hardware)*.
> There are **3 sets of MIPS instructions**: R-Type, I-Type (load/store), and J-Type (branch)
>
> - R-Type: `add, sub, and, or, slt`
> - I-Type: `lw, sw`
> - J-Type: `beq, j`
>
> PC: **P**rogram **C**ounter, aka target address or `PC + 4`

2. Definitions

2.1. ALU

For each instruction, depending on its class, the Arithmetic-Logical Unit (**ALU**, a combination of circuit used for computing a variety of arithmetic and logical functions) calculates different *arithmetic results*, *memory addresses* (for load & store, branch).

2.2. 5 Stages of Instruction

There are 5 stages that each instruction goes through:

1. **IF**: Instruction fetch from memory
2. **ID**: Instruction decode & register read
3. **EX**: Excute operation or calculate address
4. **MEM**: Access memory operand
5. **WB**: Write result back to register

2.3. Processor Elements

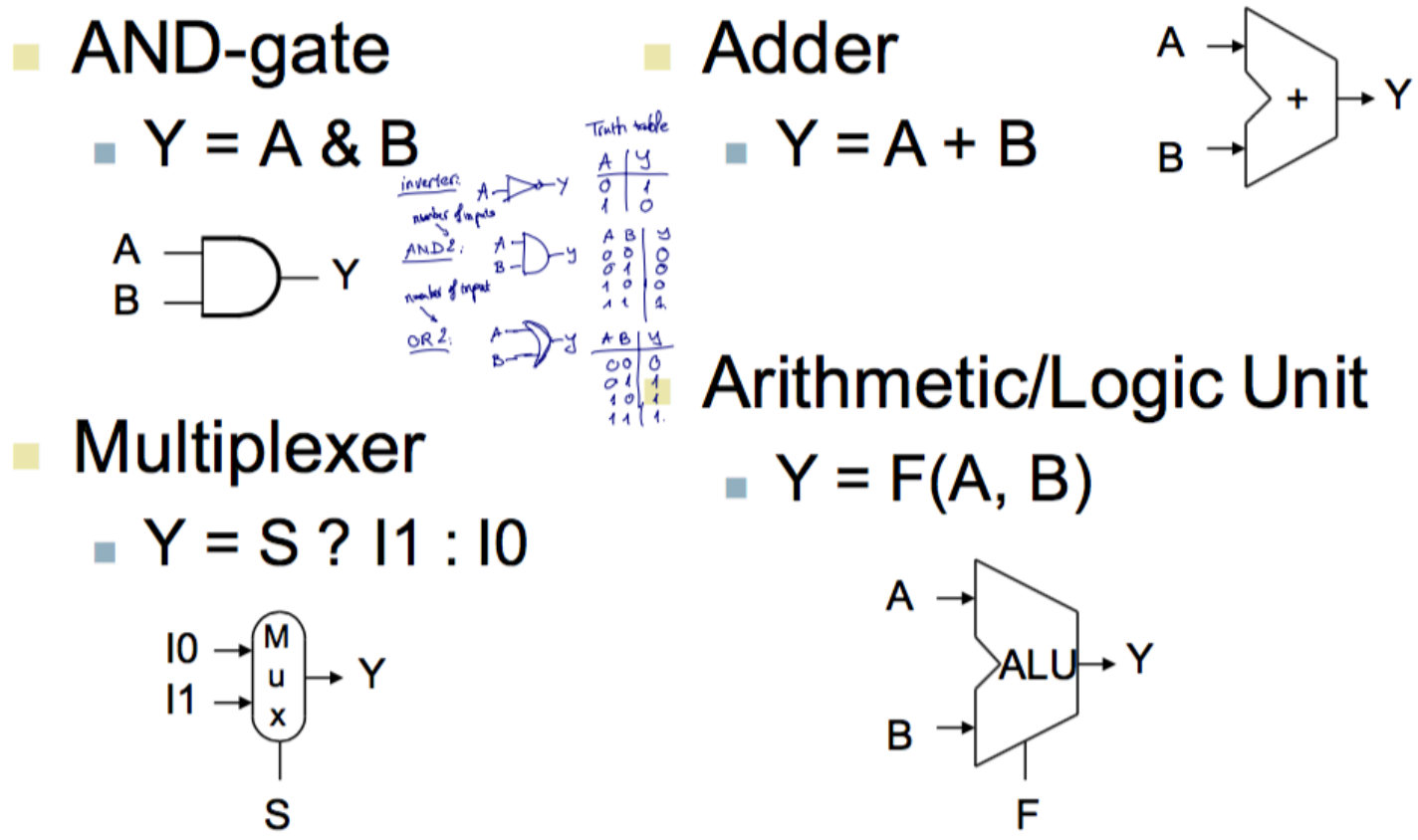
Information is encoded in binary, 1 wire conveys 1 bit, so multi-bit data uses multi-wire buses.

2.3.1. Elements

2.3.1.1. Combination element

> Output is the function of input *(i.e. input change triggers output change)*, **does not have clock nor feedback circuit**.

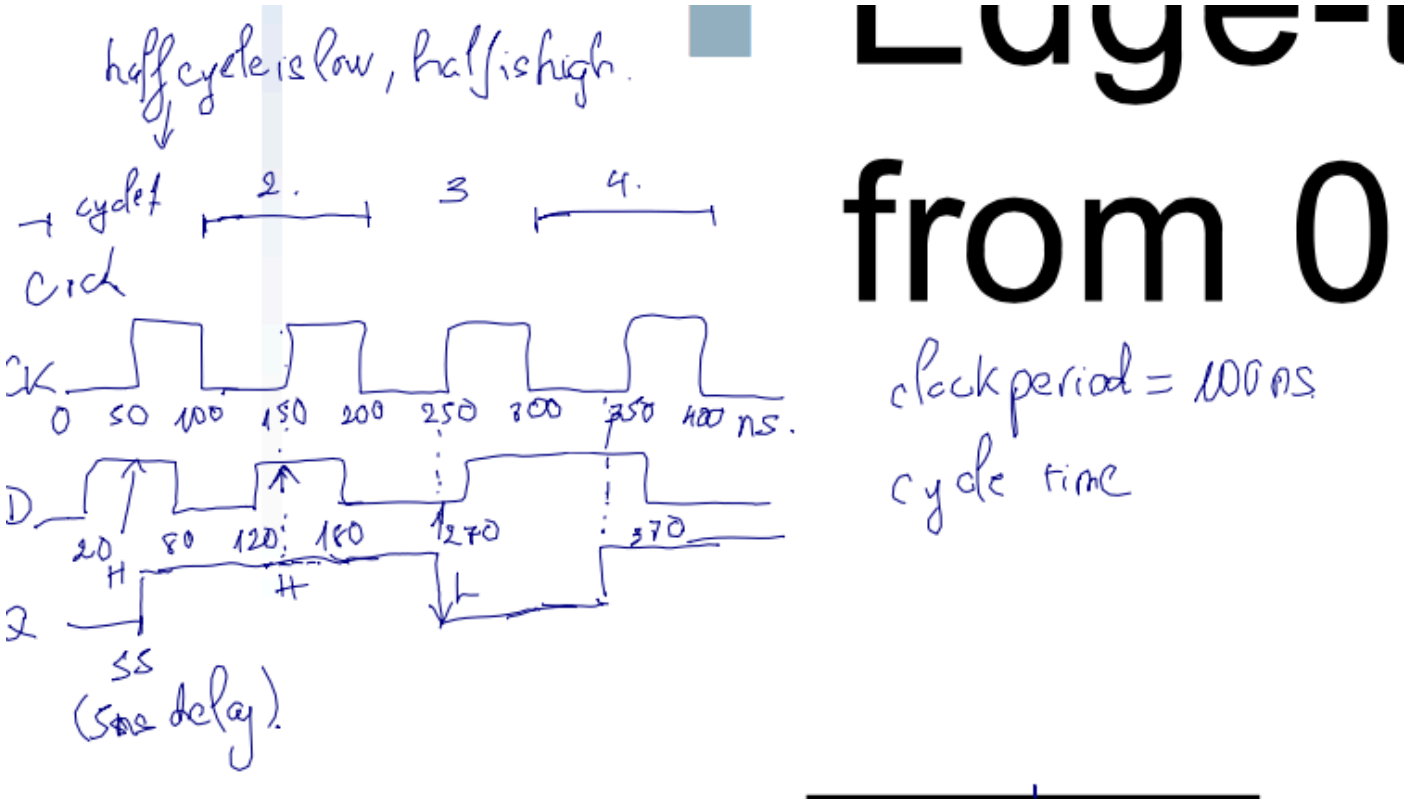
Example: AND-gate, Adder, Multiplexer, ALU (Arithmetic Logic Unit)



2.3.1.2. State element (aka sequential element)

> Stores data in circuit, output depends on both input and previous output (i.e. feedback). It relies on **clock**

> **& feedback circuit** to differentiate old/new output.
> It uses the clock signal determines when to update the stored value (when clock changes from 0 to 1).
> Data is transformed between clock edges, input from state elements, output to state element.
> Example:

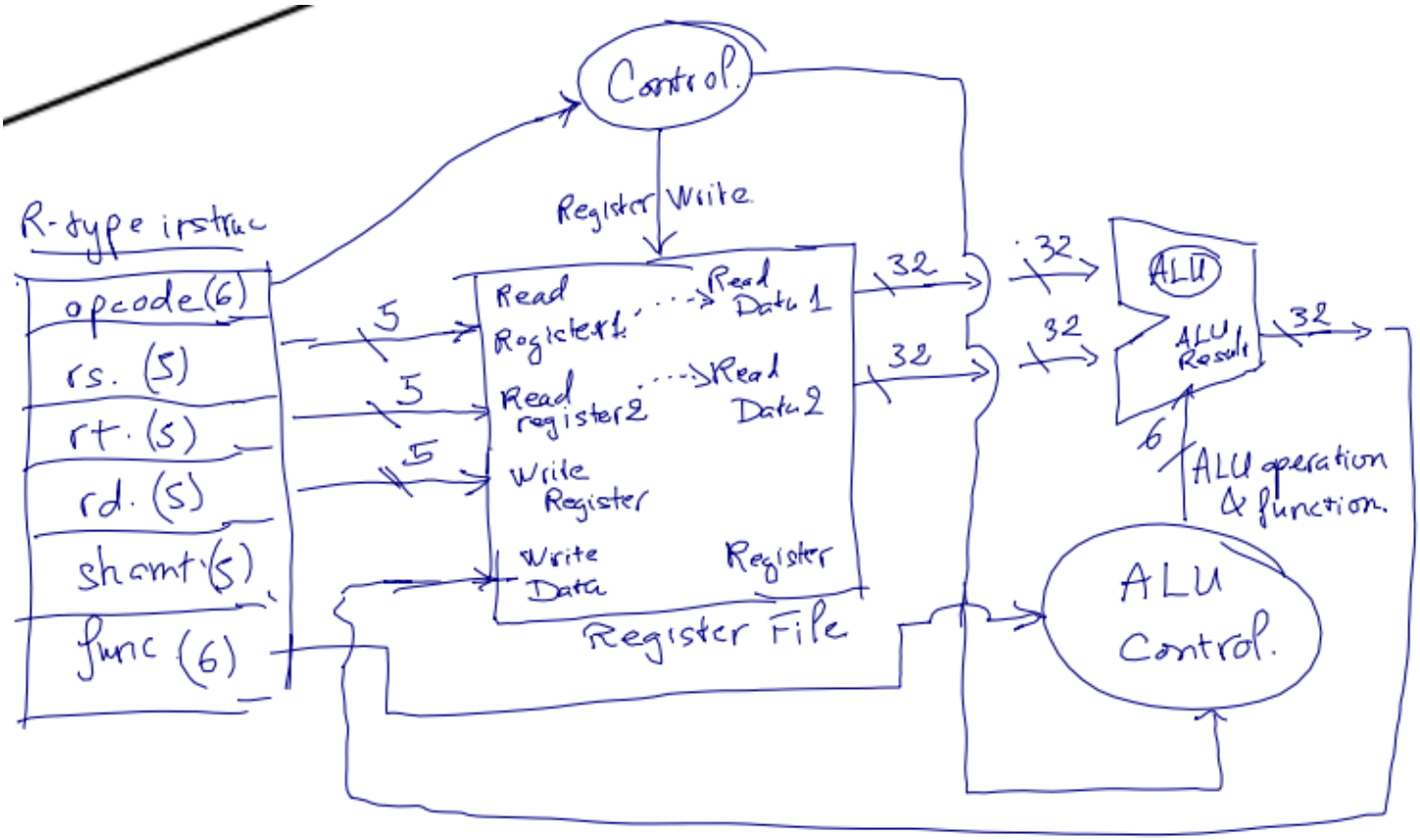


2.3.2. Datapath

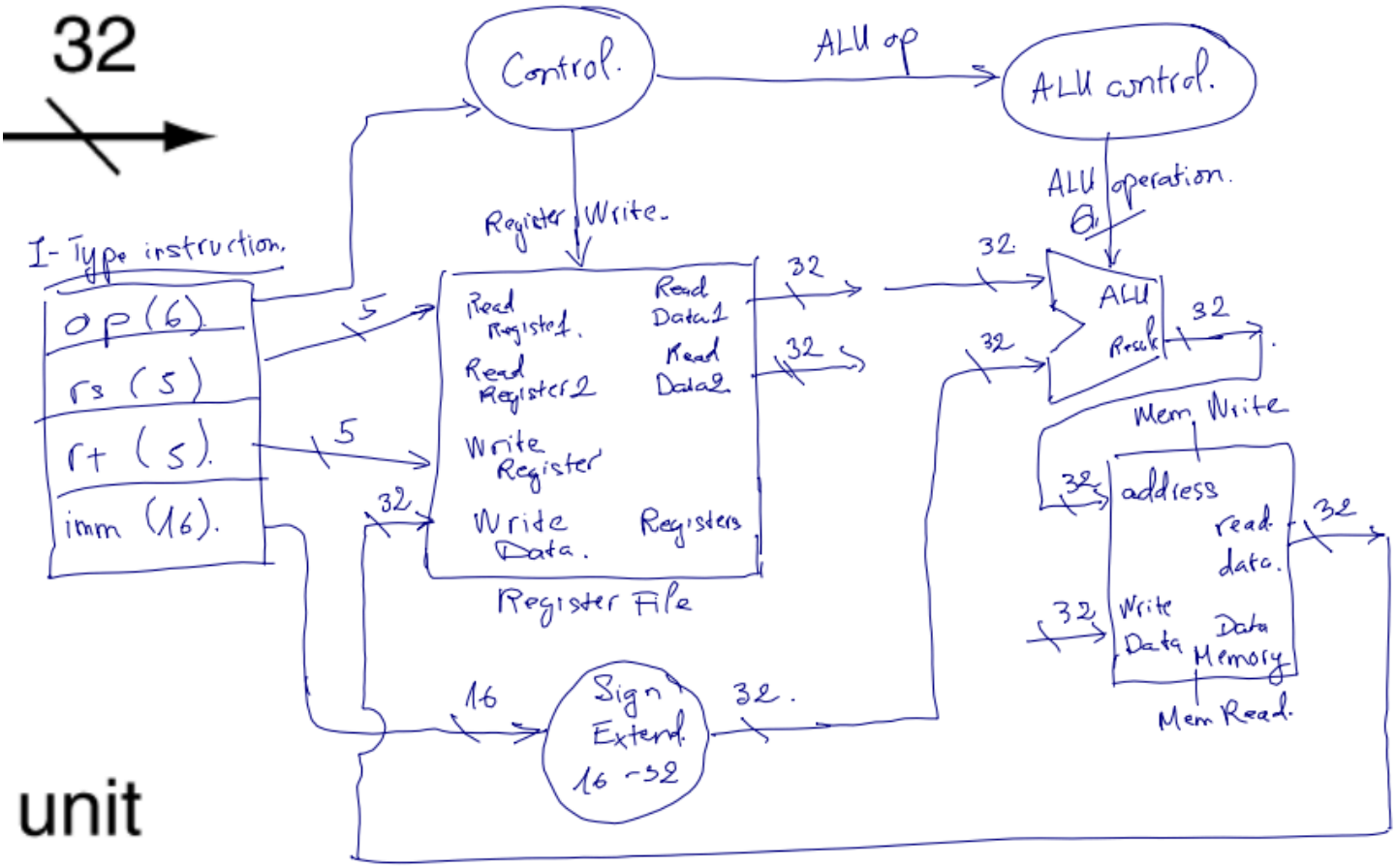
> Elements that process data and address in the CPU. Typical datapath is
> **Instruction Memory** *(read instr content)* –> **Register File** *(read data)* –> **ALU** *(compute)* –> **Data Memory** *(load some additional data/store computed data)* –> **Register File** *(write result)*`.

- Datapath with R-Format Instruction: reads 2 register operands (`rs, rt`), perform arithmetic logical operation and write result to register `rd` .
- Datapath with I-Type Instruction (load/store): read register operands, calculate address by ALU and sign-extend of offset (16-32). If LOAD: read memory from address and update register, if STORE: write register value to memory address.
- Datapath with J-Type Instruction (branch): read register operands, compare using ALU, subtract and check ZERO output, then calculate target address by sign-extend displacement, word displacement (shift left 2 places), add `PC+4`

Sample R-Format DataPath



Sample I-Format DataPath (`lw`)



Jump uses *word address*, and update `PC = top_4_bit(old_PC) + jump_address (26-bit) + 00`

Multiplexer

> is used where alternate data sources are used for different instructions.

2.4. Performance

Performance of datapath is determined by the longest delay, which is usually the load instruction stage. The performance can be improved by **pipelining**, which increases total throughput, reduce average job time, but do not change absolute job time.

$$TimeBetweenInstructions_{pipelined} = \frac{TimeBetweenInstructions_{non-pipelined}}{NumberOfStages}$$

, if all stages are balance

2.5. Hazards

> Situations that prevent starting the next instruction in the next cycle. It would cause pipeline "bubble" (wait for x cycle until condition is met).
> There are 3 types of hazard:

- structure hazards *(resource busy)*, uses separate instr/data cache.
- data hazard *(wait for read/write from prev command)*, uses theing - aka bypassing - not always work.
    - *bypassing*: use the result right after it is computed, before storing to a register (requires extra connections in datapath)
- control hazard aka branch hazard *(control action depends on previous instruction)*, uses code reordering to avoid stall. Branching hazards (also known as control hazards) occur with branches. The processor will not know the outcome of the branch when it needs to insert a new instruction into the pipeline (normally the fetch stage).