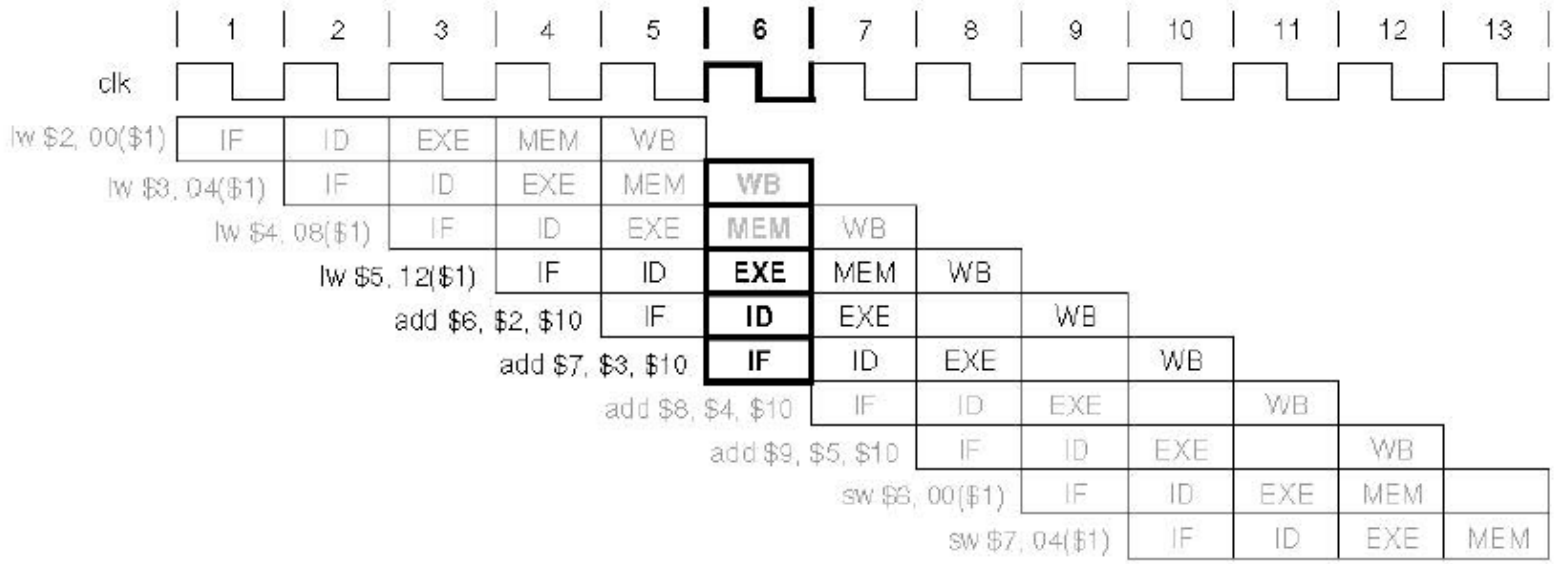
Lab 3 – Pipelining in CPU Design

Yuanqing Miao

https://github.com/myuanqing/CMPEN331_Spring2025

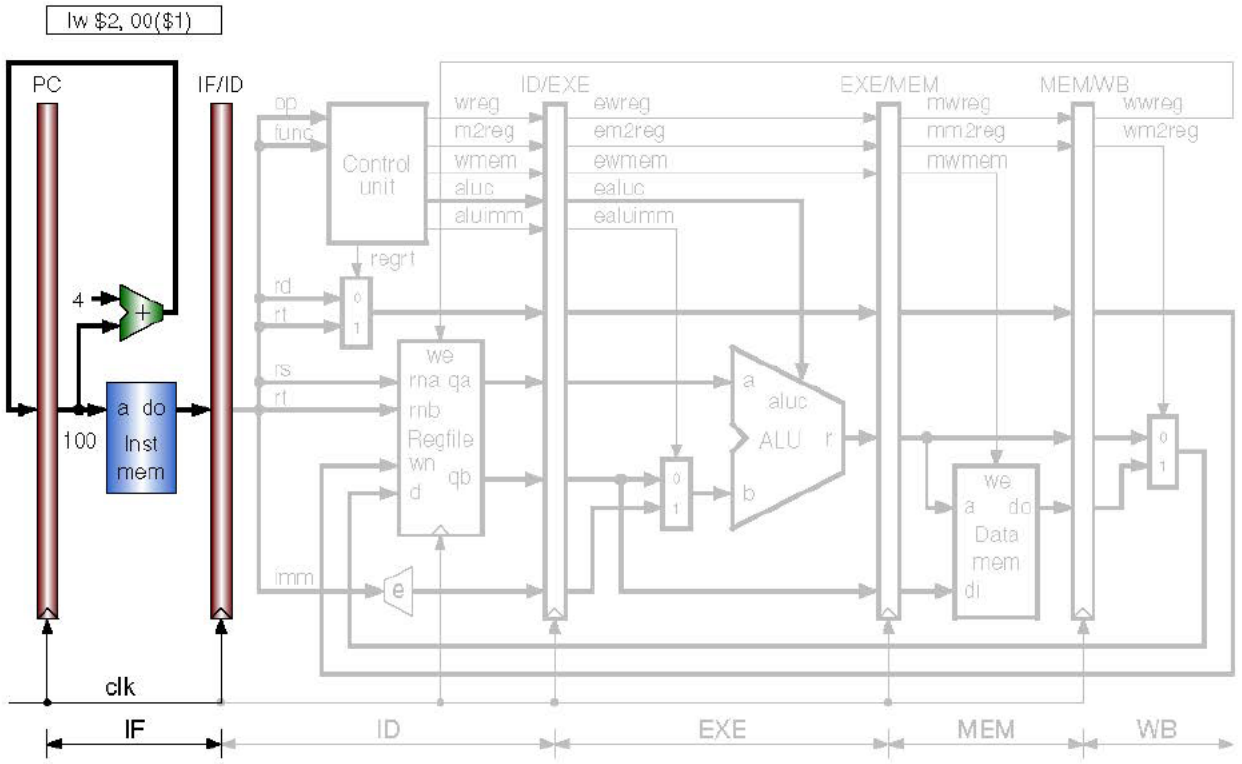
Introduction to Pipelining

- Definition: Technique to overlap execution of multiple instructions
- Five stages: **IF, ID**, EXE, MEM, WB
- Goal: Produce a result every clock cycle

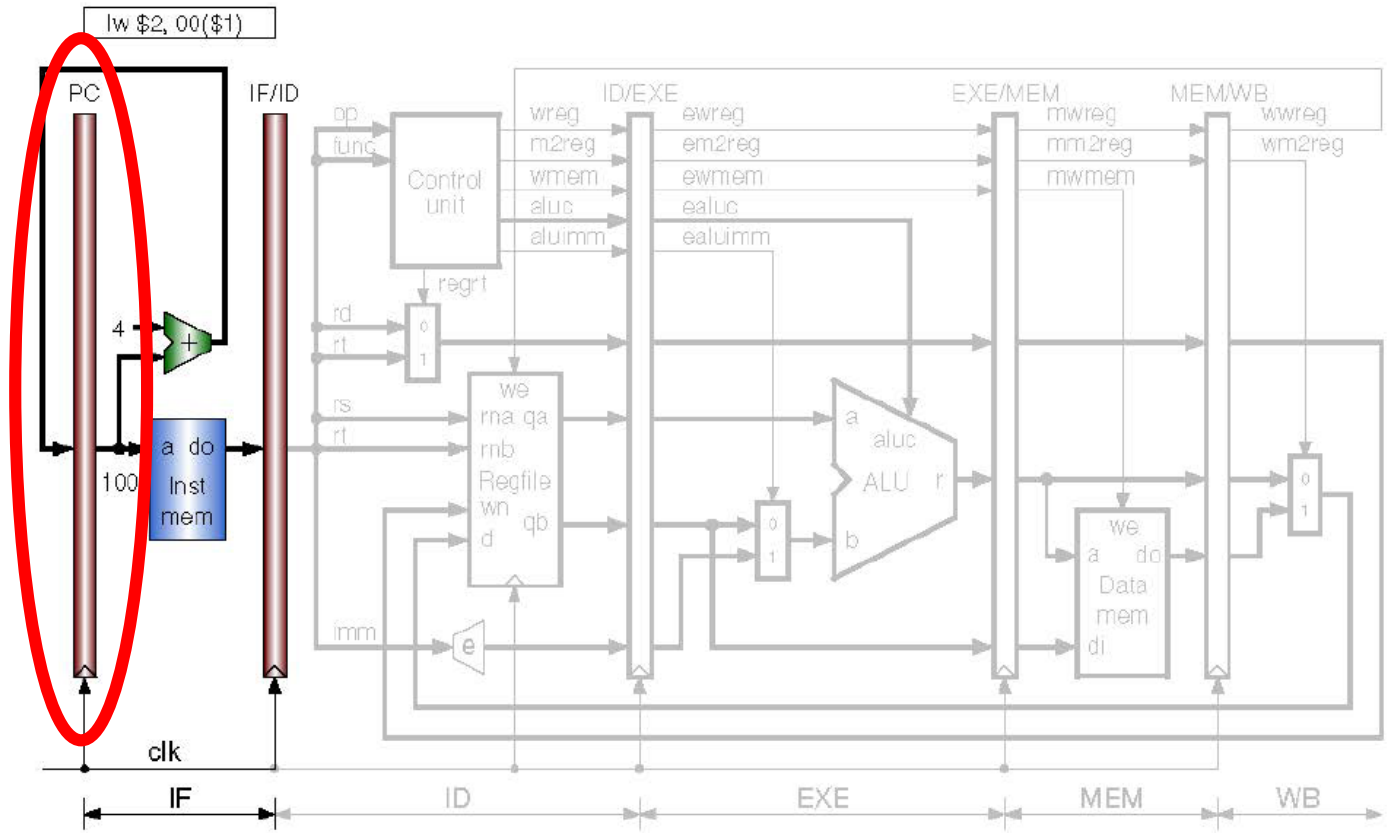


Instruction Fetch (IF) Stage

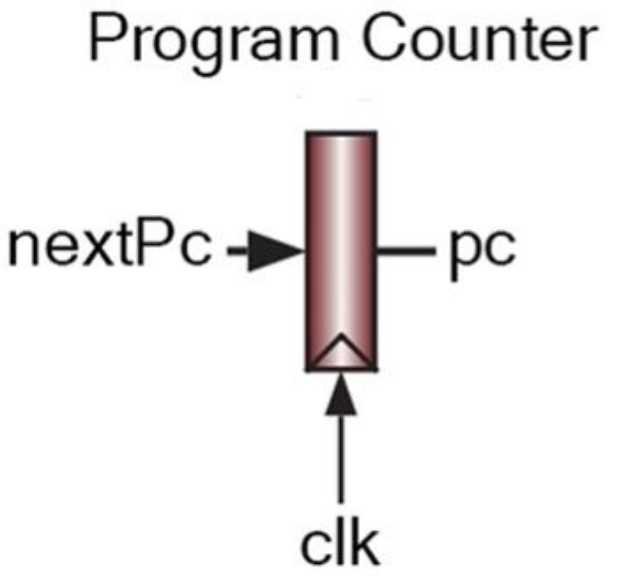
- Components: Instruction memory, adder, PC register
- Operation: Fetch instruction and calculate next PC



Module 1: PC


lw $2, 00($1)
PC
IF/ID
ID/EXE
EXE/MEM
MEM/WB
4
100
a do
Inst
mem
op
func
Control
unit
regrt
wreg
m2reg
wmem
aluc
aluimm
ewreg
em2reg
ewmem
ealuc
ealuimm
mwreg
mm2reg
mwmem
wwreg
wm2reg
rd
rt
rs
rt
we
rna qa
rnb
Regfile
wn qb
d
imm
e
a
aluc
ALU
r
b
we
a do
Data
mem
di
clk
IF
ID
EXE
MEM
WB

Module 1: PC



Program Counter

Inputs:

- nextPc [32 bits]
- clock

Output:

- pc [32 bits]

Initial Values:

- pc should be set to 100 in an initial statement

Functionality: at positive edge of clock:

- pc is set to value of nextPc

Module 1: PC

Program Counter

Inputs:

- nextPc [32 bits]
- clock

Output:

- pc [32 bits]

Initial Values:

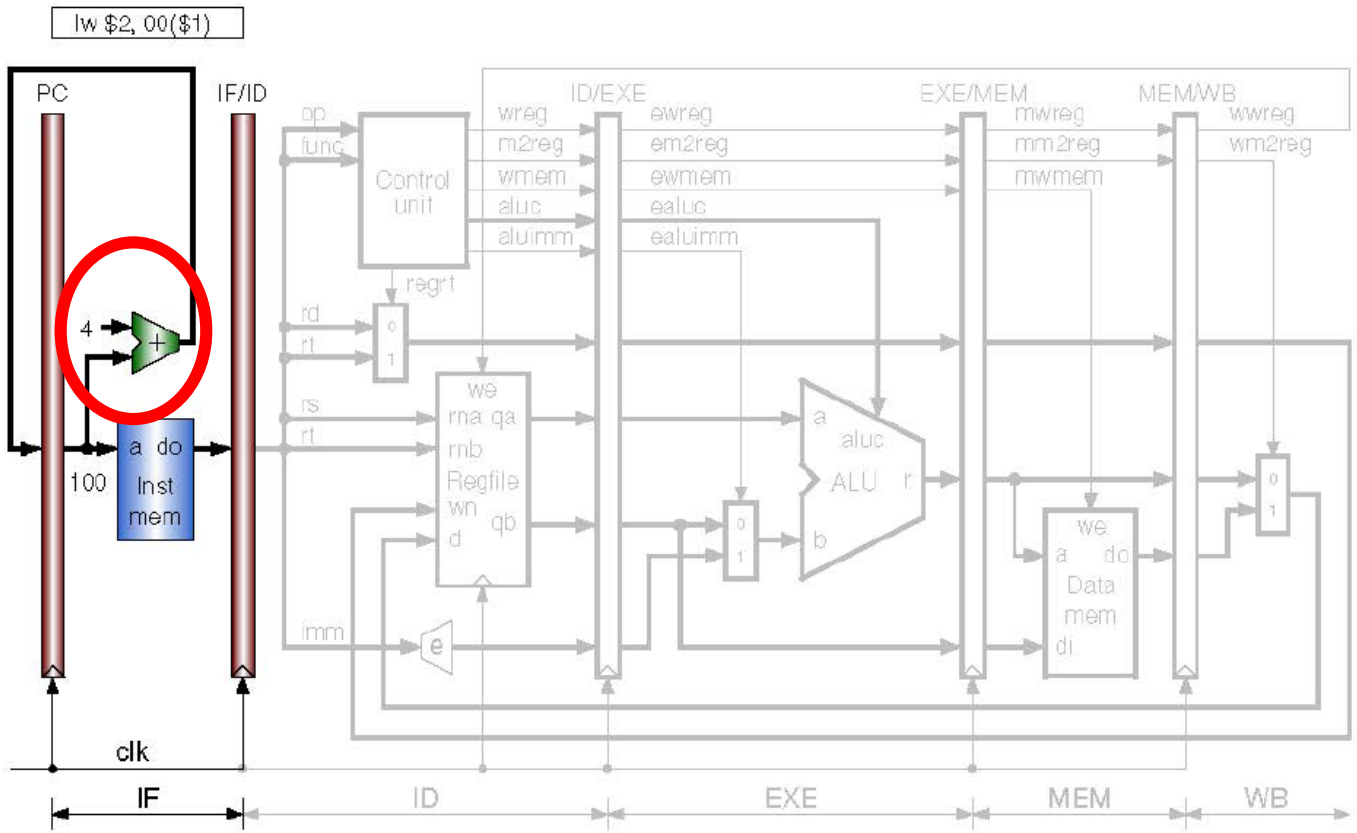
- pc should be set to 100 in an initial statement

Functionality: at positive edge of clock:

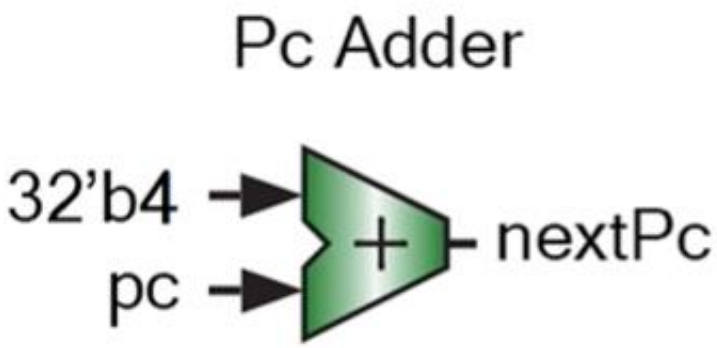
- pc is set to value of nextPc

```
23  module PCRegWrite(clock,in,out);
24  input [31:0] in;
25  input clock;
26  output reg [31:0] out;
27
28  initial begin
29   out=32'd100;
30  end
31
32  always @(posedge clock) begin
33      out=in;
34  end
35  endmodule
```

Module 2: PC Adder



Module2: PC Adder



Pc Adder

Inputs:

- pc [32 bits]
- hard-wired value of 4 [32 bits]
  - Note that this hard-wired 4 can be implemented internally and not exposed as an input if you would find this easier

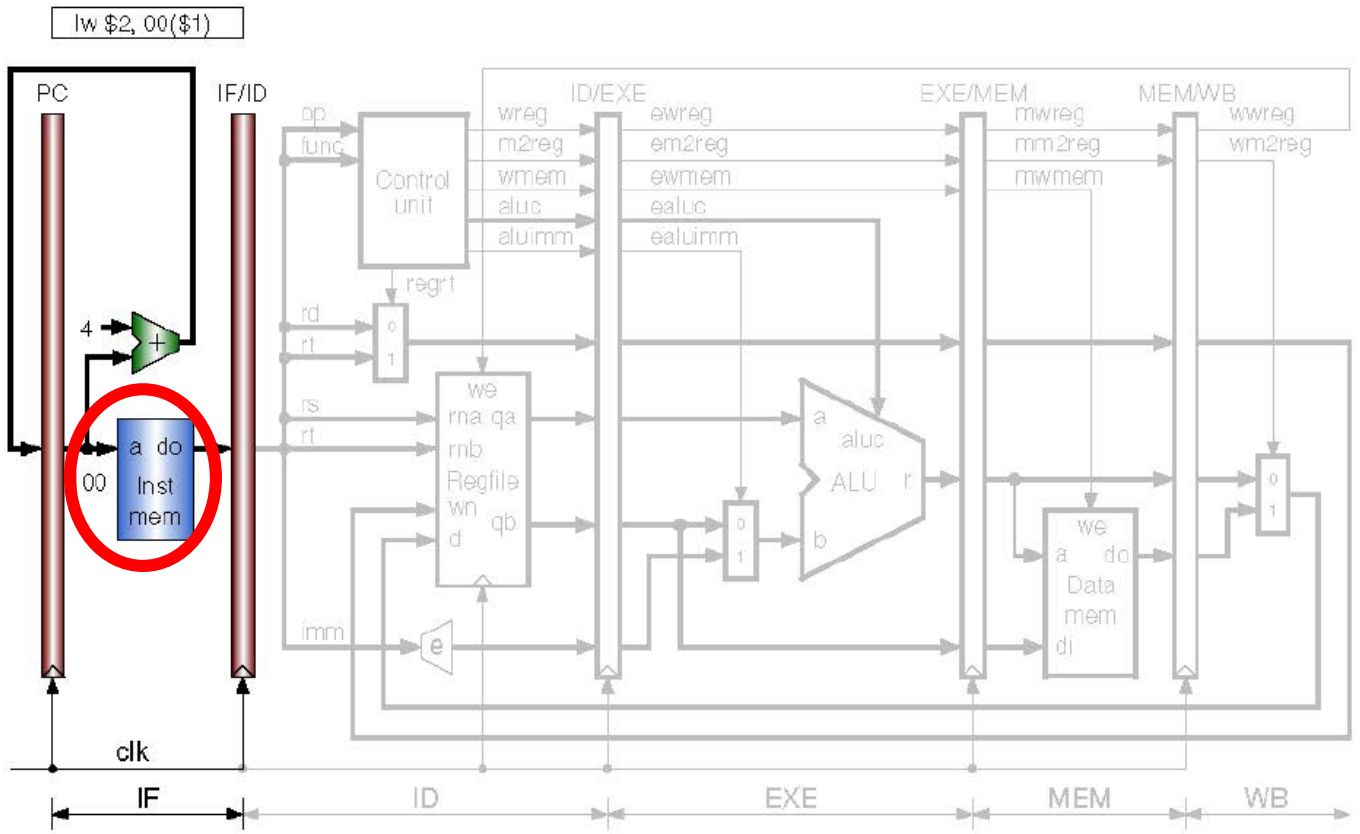
Output:

- nextPc [32 bits]

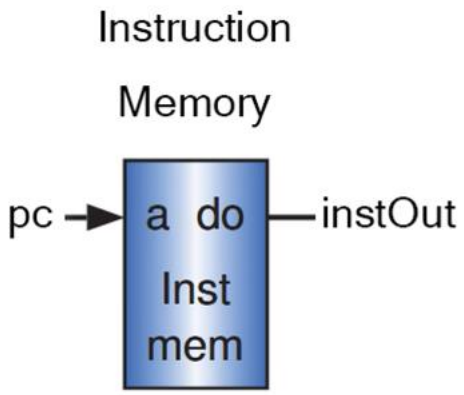
Functionality: on any signal change:

- nextPc is set to pc + 4

Module 3: Instruction Memory



Module 3: Instruction Memory



Instruction Memory

Input:

- pc [32 bits]

Output:

- instOut (instruction out) [32 bits]

Internal Parts:

- memory (implemented as 2D reg array, more on this on the next few slides)

Functionality: on any signal change:

- instOut is set to value of memory array at position pc

Module 3: Instruction Memory

- Normally, memory in MIPS is byte-addressed, meaning each address points to a single byte of data.
- However, to simplify for our implementation, we'll make our memory store whole words (4 bytes) of data and only read/write on 4-byte boundaries in order to simplify our instruction memory and, in lab 4, our data memory.

- For our implementation, we'll need a minimum of 256 bytes, or 64 words. Since we'll only read from instruction memory on 4-byte boundaries, we can implement our memory as a 32x64 block (32-bit values, 64 different possible positions).
- To create a 32x64 register array in Verilog, we do the following:
  - ```
    reg [31:0] memory [0:63];
    ```
    ```

Module 3: Instruction Memory

- We can access a word in memory stored at the value of our program counter in our Verilog implementation by doing the following:
  - `instOut = memory[pc[31:2]];`
- We can also simplify this further. Since we only have 64 words in memory, we only need 6 bits to address all the values in data memory ($2^6 = 64$). We can then write the Verilog as follows:
  - `instOut = memory[pc[7:2]];`
- Note that this implementation will cause issues if we try to run a program longer than 64 words/instructions, but we won't ever do that in this course.

Module 3: Instruction Memory

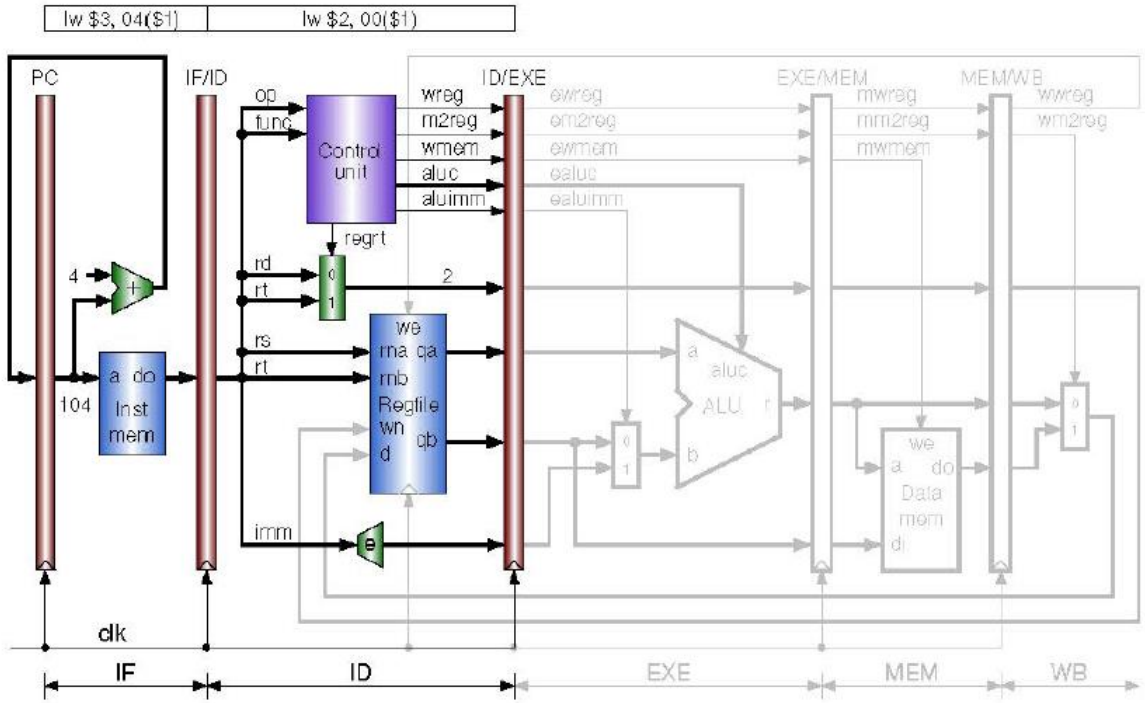
- We'll need to initialize some values into our instruction memory. To do this, we need to add an initial begin block to your InstructionMemory module, and manually assign these values.

- We can assign values to memory by doing the following in our initial block:
  - `memory[25] = 32'h00000000;`

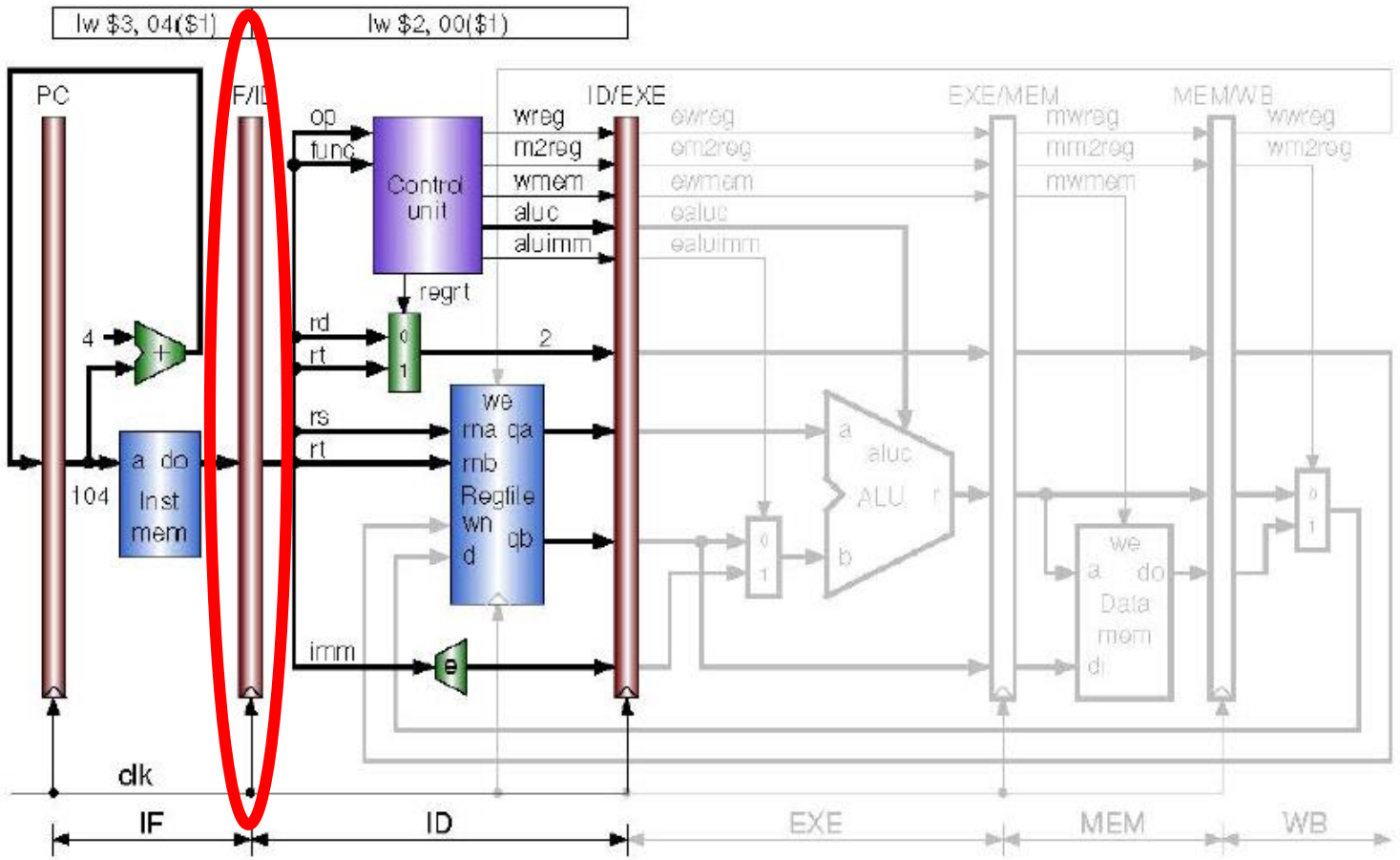
- You only need to initialize the values of memory where instructions are located (for lab 3, this is words 25 and 26). You do not need to zero-fill the rest of memory, you can leave it uninitialized, which defaults to don't-cares in simulation.

Instruction Decode (ID) Stage

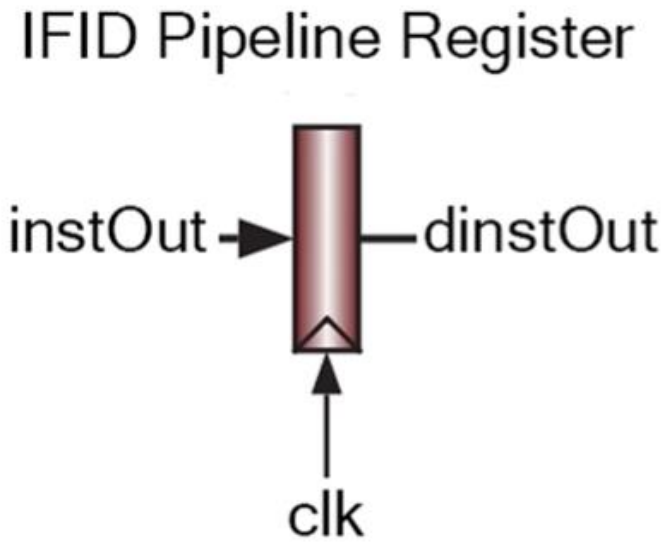
- Purpose: Decode instruction and read operands
- IF/ID Register, Control Unit, Register Multiplexer, Register file, Immediate Extender, IDEXE Pipeline Register



Module 4: IFID Pipeline Register



Module 4: IFID Pipeline Register



Ifid Pipeline Register

Inputs:

- instOut [32 bits]
- clock

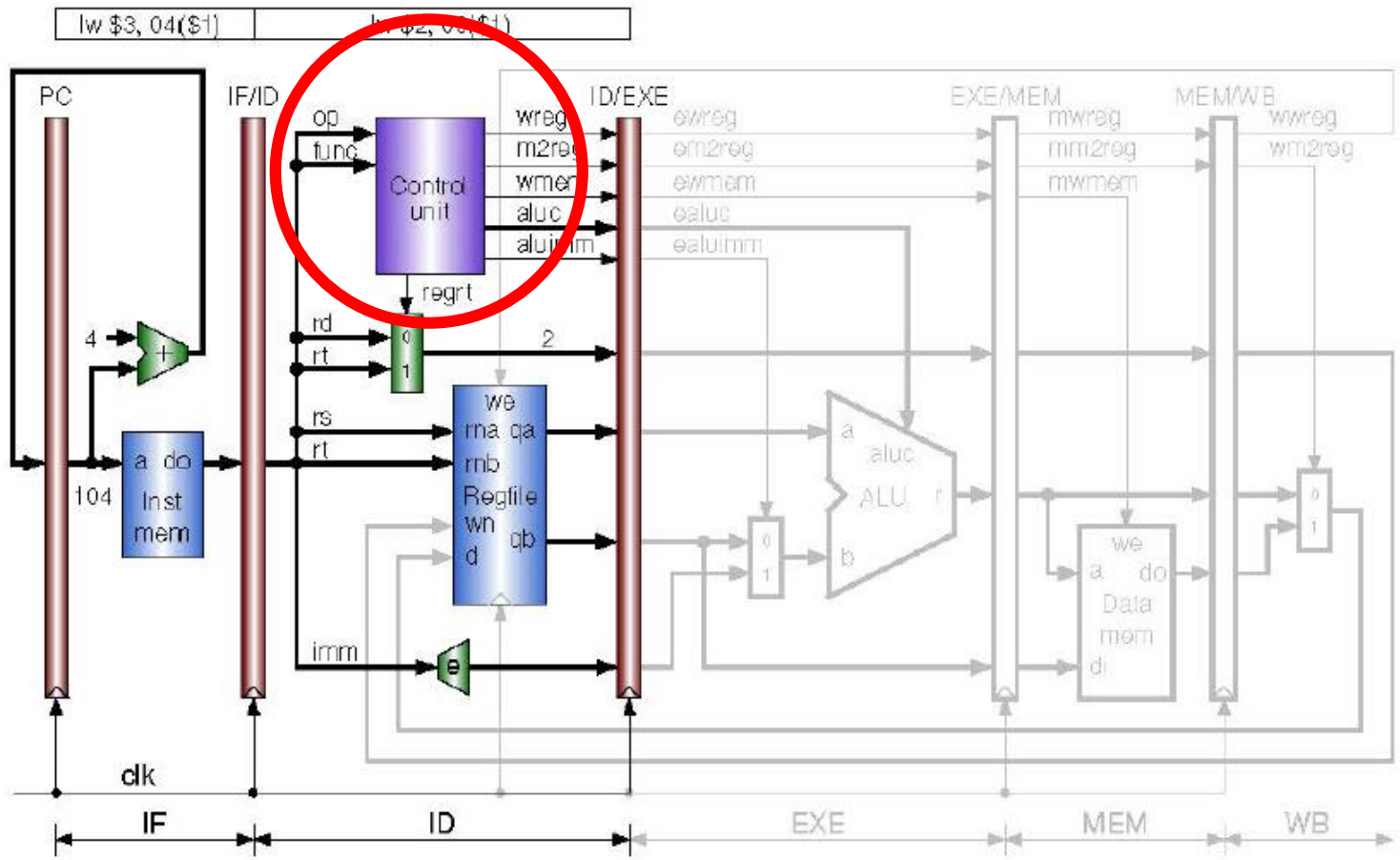
Output:

- dinstOut [32 bits]

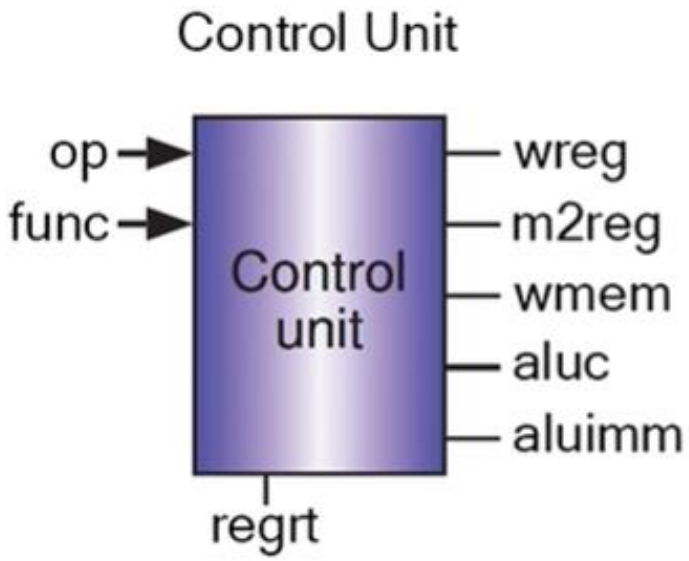
Functionality: at positive edge of clock:

- dinstOut is set to instOut

Module 5: Control Unit



Module 5: Control Unit



Control Unit

Inputs:

- op (bits 31:26 of dinstOut) [6 bits]
- func (bits 5:0 of dinstOut) [6 bits]

Outputs:

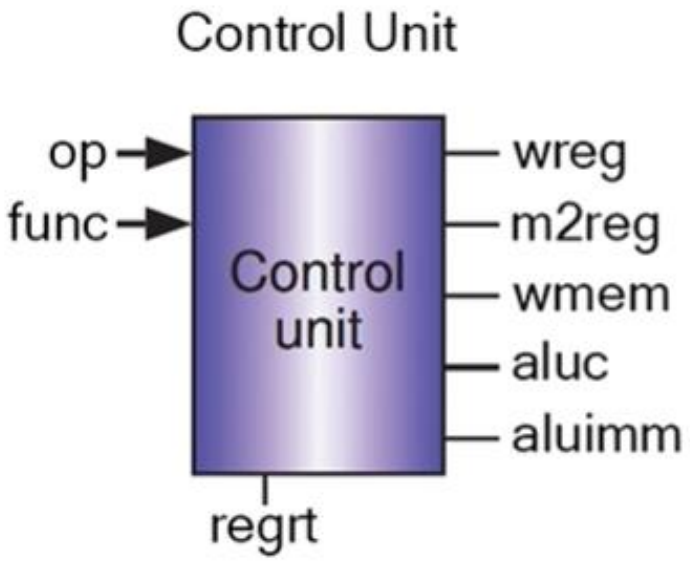
- wreg
- m2reg
- wmem
- aluc [4 bits]
- aluimm
- regrt

Functionality: on any signal change:

- all outputs are assigned based on the values of op and func (more on this on next few slides)

Module 5: Control Unit

- **wreg:** Write to the register file
- **m2reg:** Load memory value to a register
- **wmem:** Write to data memory
- **aluc:** ALU control (add operation = 2)
- **aluimm:** Use sign-extended immediate
- **regrt:** Select register address for writing



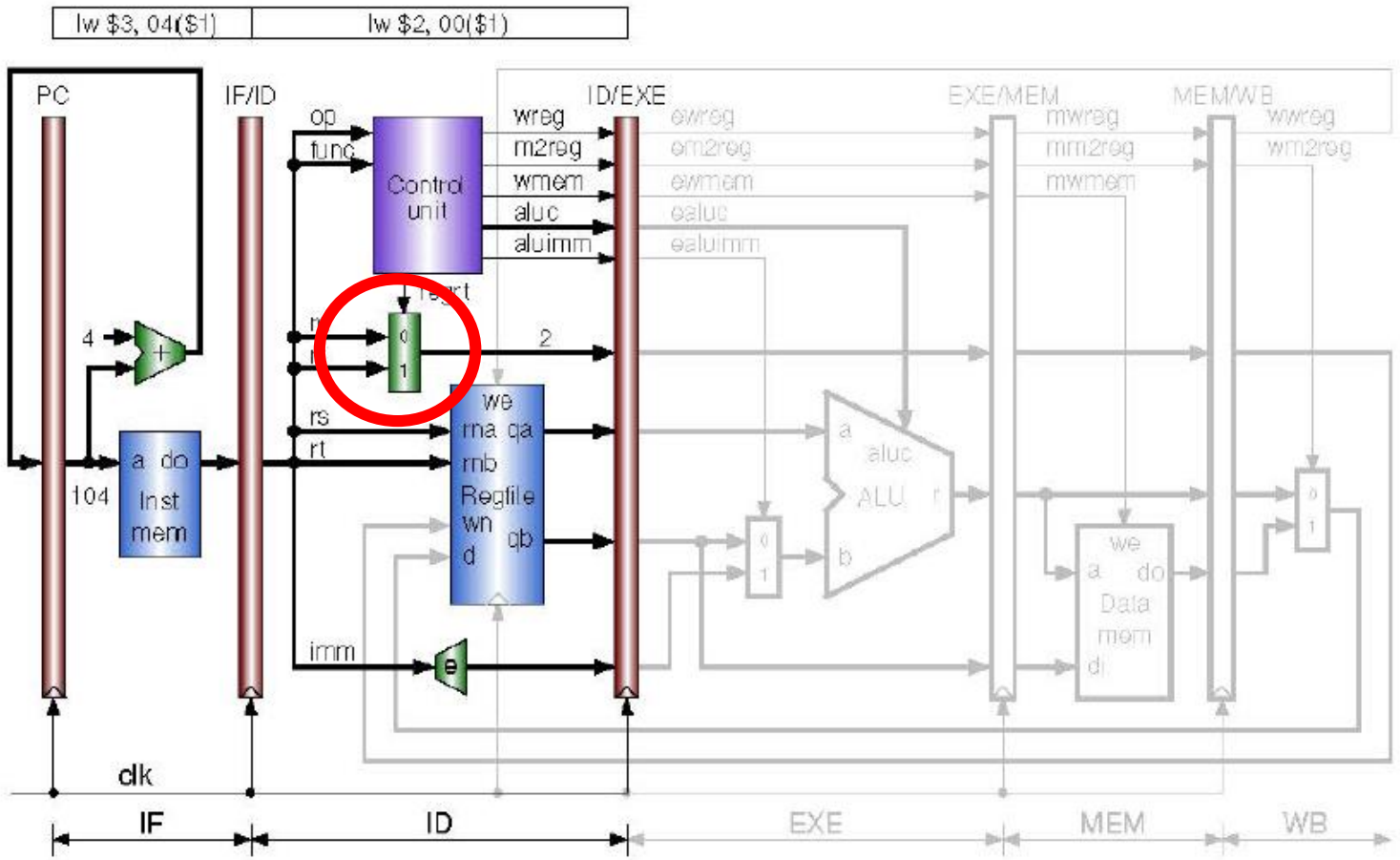
opcode	rs	rt	rd	shamt	func
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Module 5: Control Unit

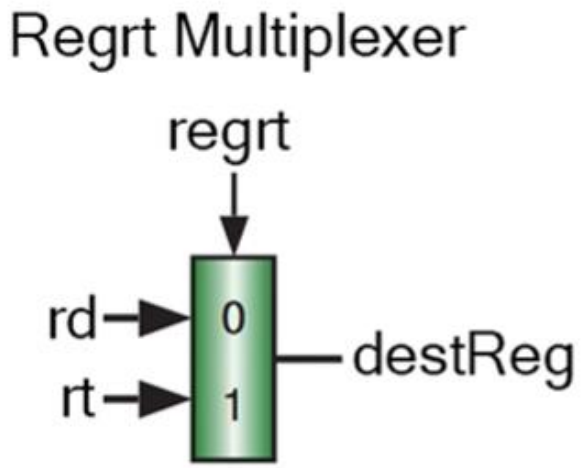
- Note that the only instructions you need to implement for the labs are Load Word and Add. Subtract is shown here to represent how R-format instructions should be handled, as you will need to implement more of the R-format instructions in the final project, but you don't need to implement Subtract for the labs.

```
always @(*) begin
    case (op)
        6'b000000: // r-types
        begin
            case (func)
                6'b100000: // ADD instruction
                    // setting values of control
                    // signals for ADD instruction
                6'b100010: // SUB instruction
                    // setting values of control
                    // signals for SUB instruction
            endcase
        end
        6'b100011: // LW
        begin
            // setting values of control signals
            // for LW instruction
        end
    endcase
end
```

Module 6: Register Multiplexer



Module 6: Register Multiplexer



Regrt Multiplexer

Inputs:

- rt (bits 20:16 of dinstOut) [5 bits]
- rd (bits 15:11 of dinstOut) [5 bits]
- regrt

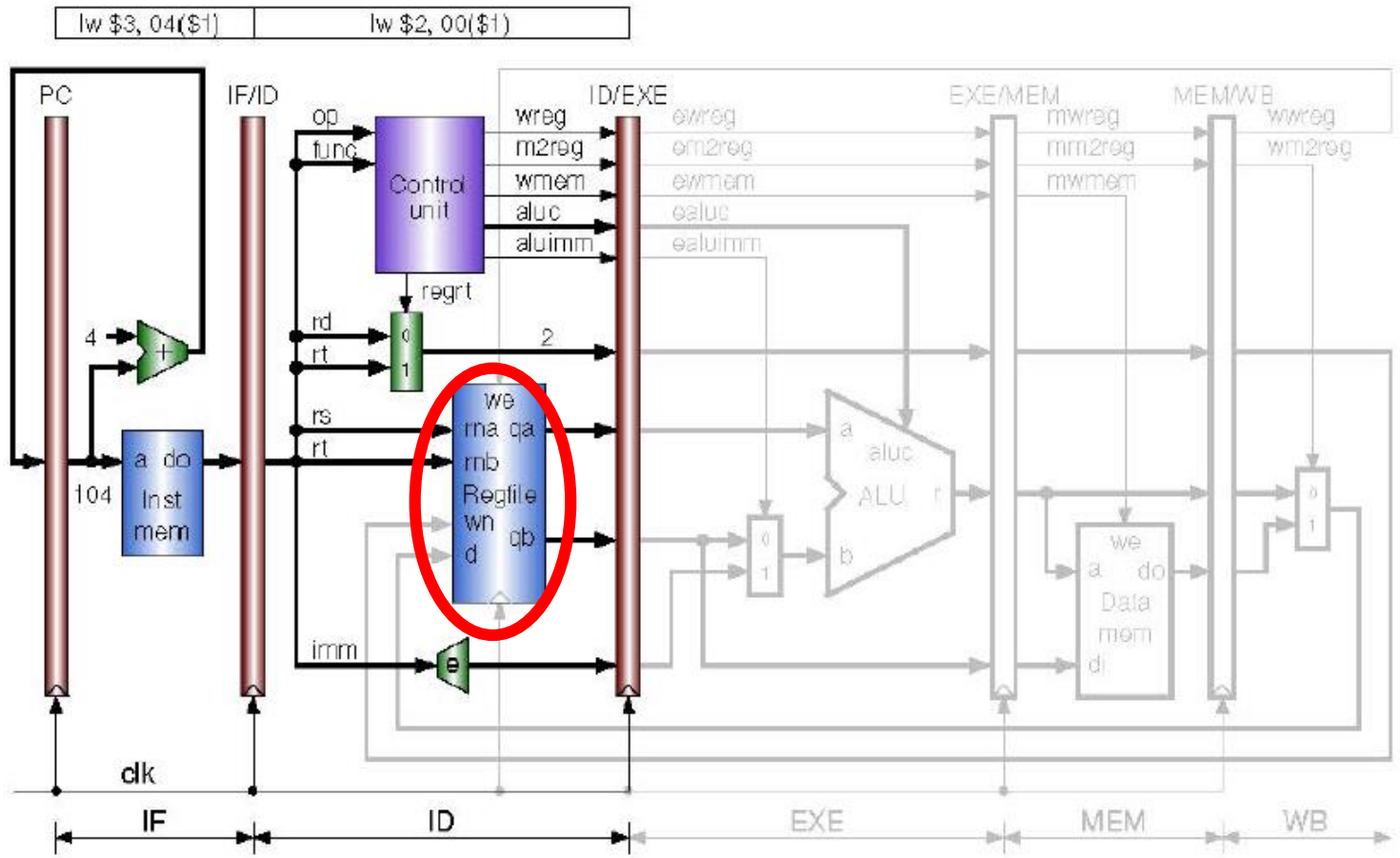
Output:

- destReg [5 bits]

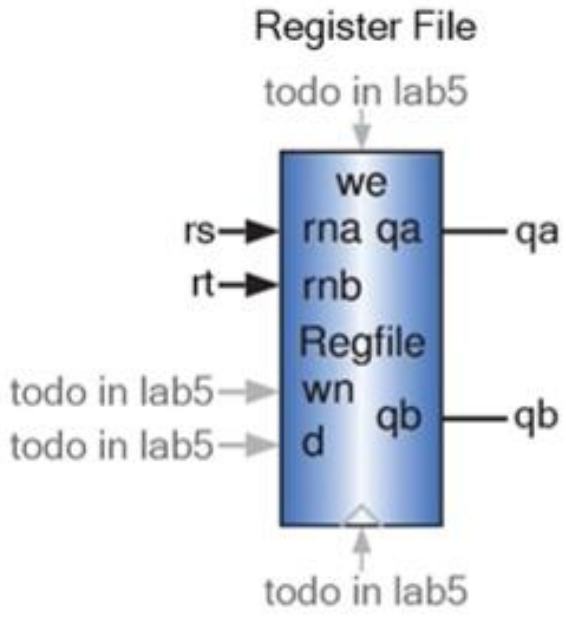
Functionality: on any signal change:

- If regrt is 0, destReg is set to value of rd
- If regrt is 1, destReg is set to value of rt

Module 7: Register File



Module 7: Register File



Register File

Inputs:

- rs (bits 25:21 of dinstOut) [5 bits]
- rt (bits 20:16 of dinstOut) [5 bits]

Outputs:

- qa [32 bits]
- qb [32 bits]

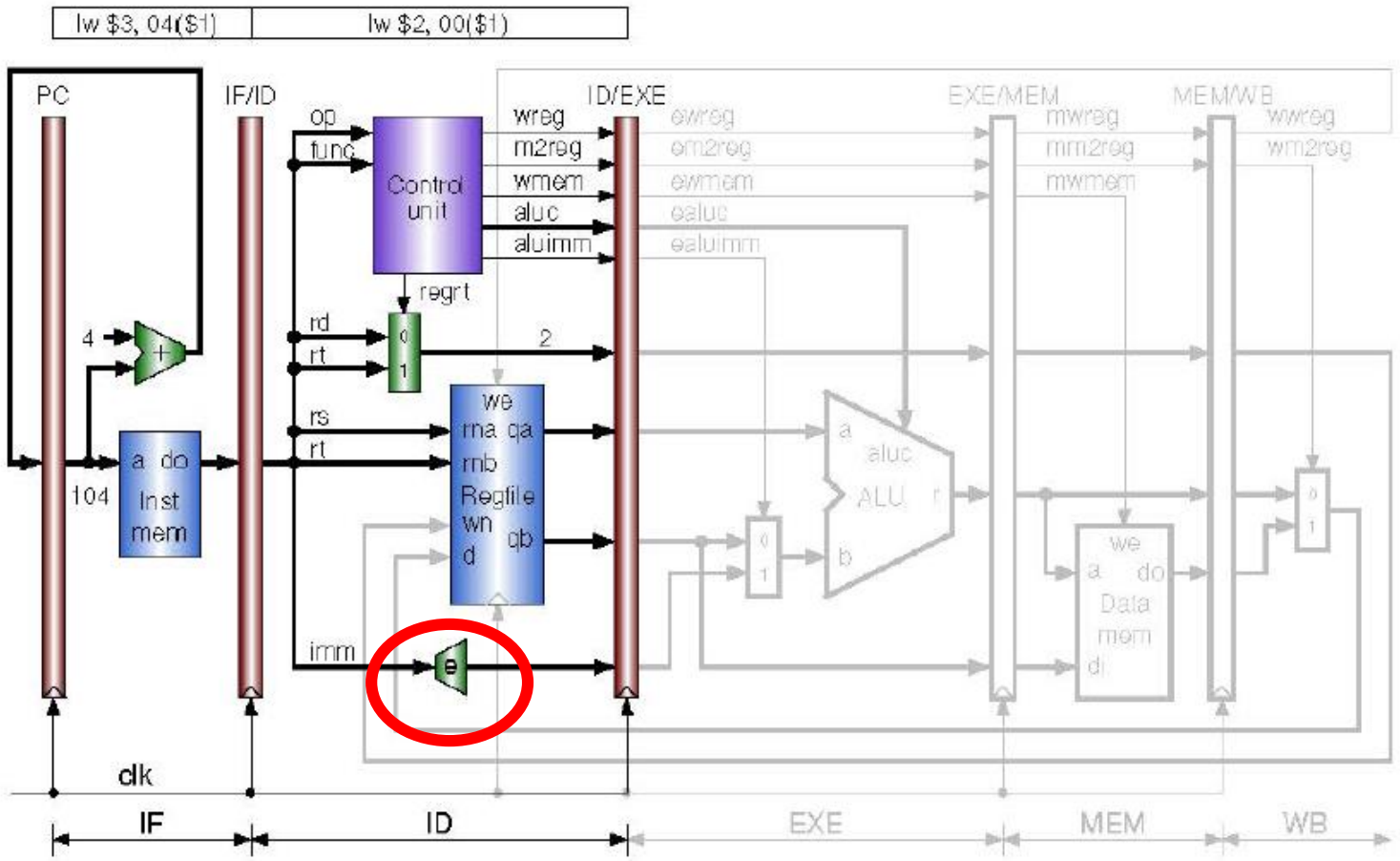
Internal parts:

- Registers (32 registers, each register is 32 bits wide, implemented as 2D reg array, more on this on the next slide)

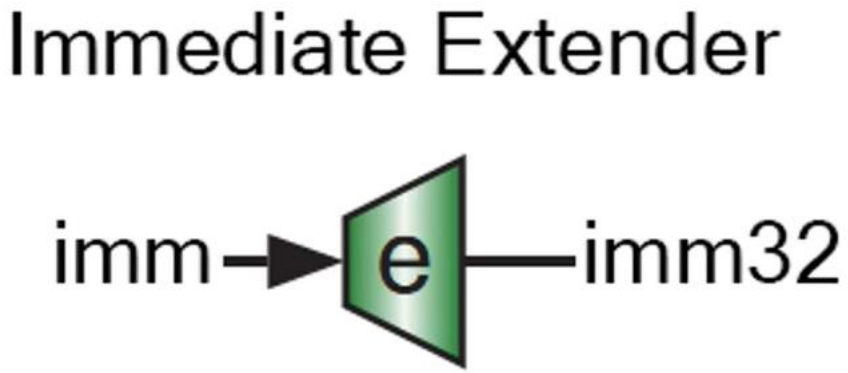
Functionality: on any signal change:

- qa is set to value of register at address rs
- qb is set to value of register at address rt
- Note: there are some additional inputs related to writing to the registers. You do not need to implement these inputs and the related functionality until lab5.

Module 8: Immediate Extender



Module 8: Immediate Extender



Immediate Extender

Input:

- imm [16 bits]

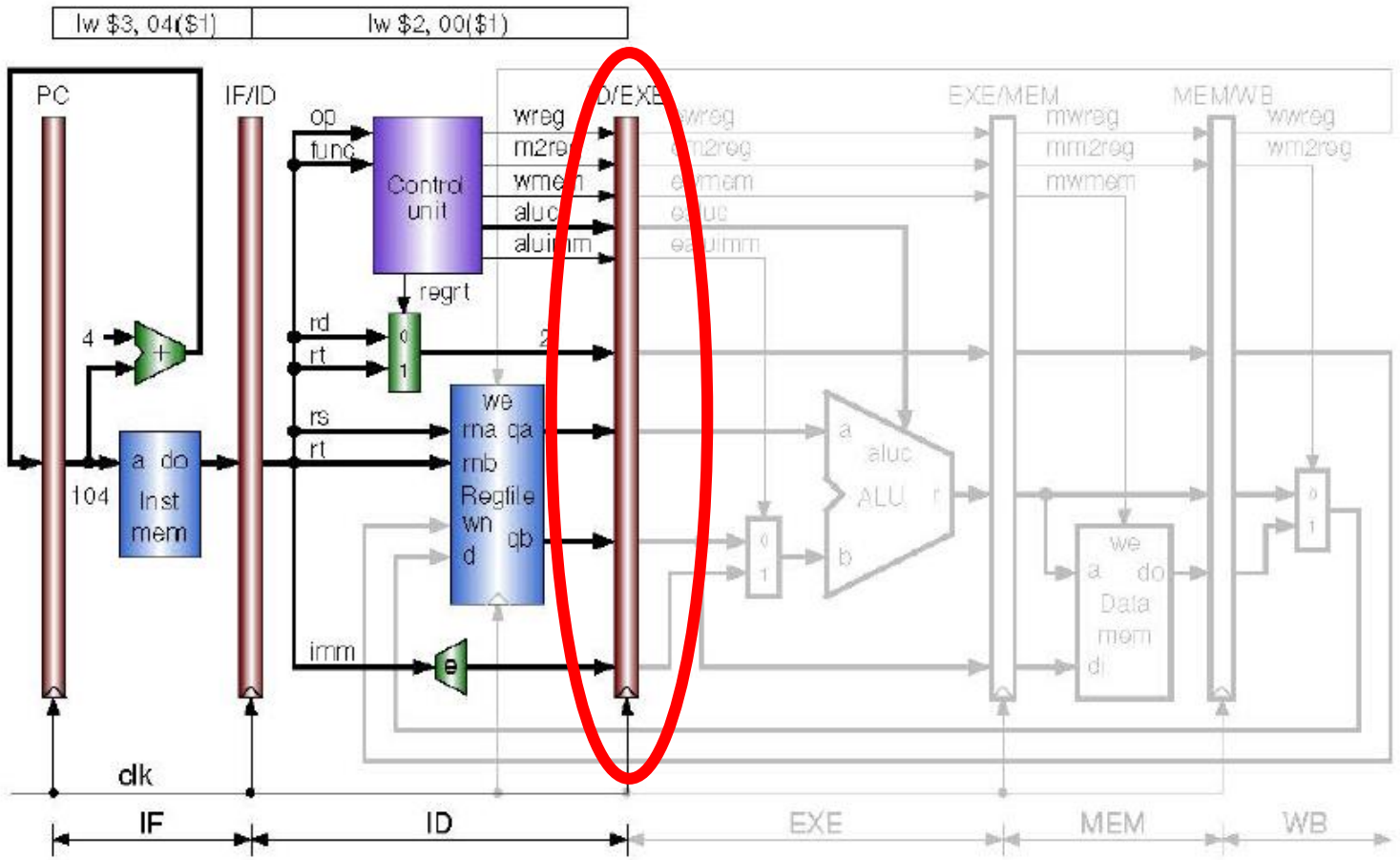
Output:

- imm32 [32 bits]

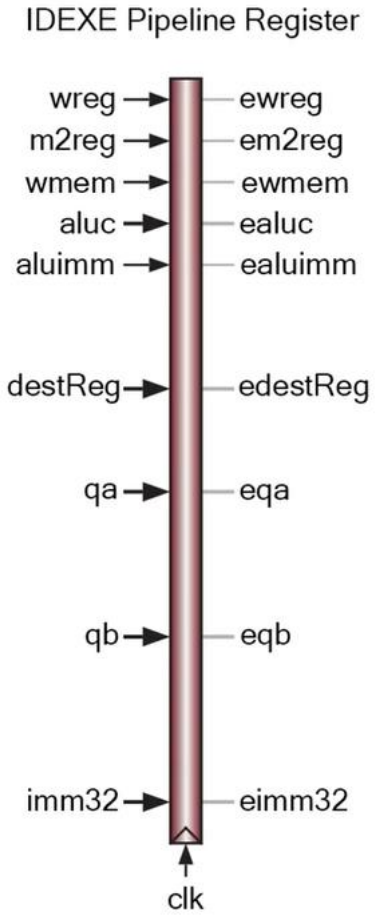
Functionality: on any signal change:

- imm32 is set to sign-extended value of imm
  - Note: sign-extension is not zero-extension! Hard wiring 16 zero bits before imm to get imm32 is incorrect!
  - Slide 60 of the Review and Verilog 1 powerpoint will help you with implementing your immediate extender.

Module 9: IDEXE Pipeline Register



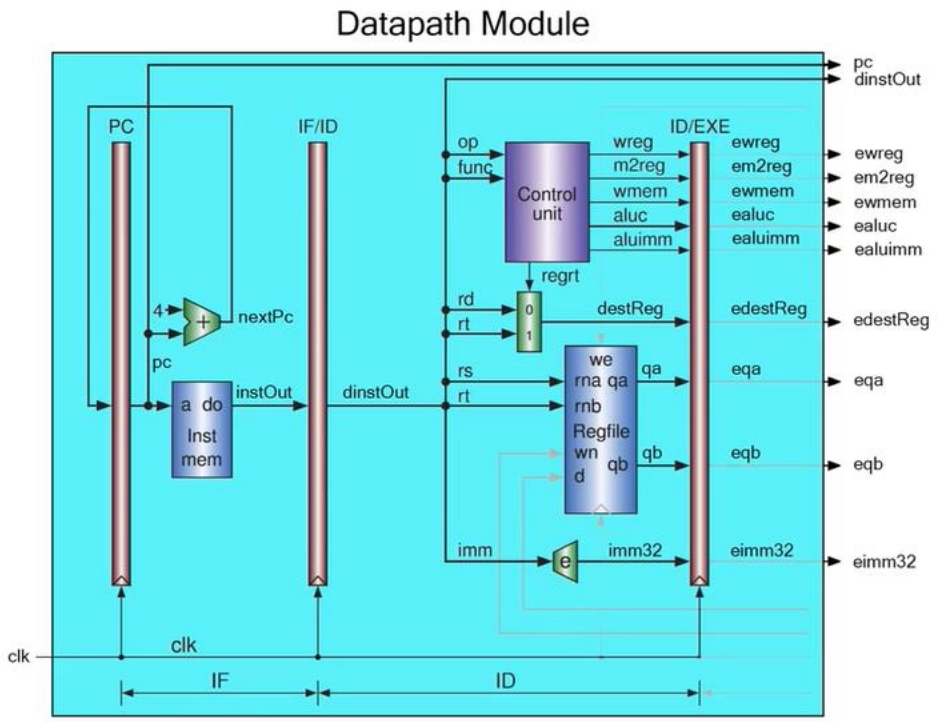
Module 9: IDEXE Pipeline Register



IDEXE Pipeline Register

Inputs:	Outputs:	Functionality: on positive edge of clock:
❖wreg ❖m2reg ❖wmem ❖aluc [4 bits] ❖aluimm ❖destReg [5 bits] ❖qa [32 bits] ❖qb [32 bits] ❖imm32 [32 bits] ❖clock	❖ewreg ❖em2reg ❖ewmem ❖ealuc [4 bits] ❖ealuimm ❖edestReg [5 bits] ❖eqa [32 bits] ❖eqb [32 bits] ❖eimm32 [32 bits]	❖ewreg set to value of wreg ❖em2reg set to value of m2reg ❖ewmem set to value of wmem ❖ealuc set to value of aluc ❖ealuimm set to value of aluimm ❖edestReg set to value of destReg ❖eqa set to value of qa ❖eqb set to value of qb ❖eimm32 set to value of imm32

Put Modules Together



Datapath

Input:

- clock

Outputs:

- pc
- All outputs from IFID and IDEXE Pipeline Registers
  - Note: all outputs in the datapath will be wires instead of registers. This is because their values are driven by the modules they are connected to, and the modules handle setting the values at the proper times when they need to be set.

Internal parts:

- The Datapath should instantiate an instance of all of the modules previously describes, as well as use wires to connect all of them together.
- The Datapath Module will not have an always block, it simply contains all of the modules needed for the CPU and connects them together, as well as exposes parts of the module outputs for the testbench.

TestBench

- **Registers:**
  - `clk`
- **Wires:**
  - `pc [32 bits]`
  - `dinstOut [32 bits]`
  - `ewreg`, `em2reg`, `ewmem`
  - `ealuc [4 bits]`, `ealuimm`
  - `edestReg [5 bits]`
  - `eqa [32 bits]`, `eqb [32 bits]`
  - `eimm32 [32 bits]`

- The testbench simply serves to generate a clock signal for the datapath, and to read all the signals coming from the pipeline registers + pc
- You'll have a clock register and the wires listed on the left in your testbench to connect to the datapath. You MUST name these wires these names, or something relatively similar. If the graders cannot tell what your wires are meant to represent in your waveforms, you may lose points.
- Note: you should initially set your clock to zero, initially setting it to one can cause funky, undesired errors/behaviors during simulation.
- **DO NOT merge your testbench and datapath into a single module, this will cause lots of errors with synthesis, viewing the schematic, and in the final project, implementation.**