*Faculty of Engineering*
*Electrical Engineering Dep.*
**Instructor: Dr.Gihan Naguib**

**CEE210/CEE216**
**Computer Engineering**
***2nd year computer/Commination***
***Spring 2021***

# Pipelined MIPS Processor Implementation

## Instruction Set Architecture

In this project, you will design a simple 32-bit RISC processor with eight 32-bit general purpose registers: R0 through R7. R0 is a normal register, NOT hardwired to zero. The program counter PC is a special-purpose 20-bit register. That can address at most $2^{20}$ instructions. All instructions are only 16 bits. There are four instruction formats, R-type, I-type, B-Type and J-type as shown below:

### *R-type format*

5-bit opcode (Op), 3-bit destination register Rd, and two 3-bit source registers Rs & Rt and 2-bit function field F

| | | | | |
|---|---|---|---|---|
| **$Op^5$** | **$Rd^3$** | **$Rs^3$** | **$Rt^3$** | **$F^2$** |

### *I-type format*

5-bit opcode (Op), 3-bit destination register Rd, 3-bit source register Rs, and 5-bit immediate

| | | | |
|---|---|---|---|
| **$Op^5$** | **$Rd^3$** | **$Rs^3$** | **$Imm^5$** |

### *B-type format*

5-bit opcode (Op), 3-bit register number Rd, and 8-bit Immediate

| | | |
|---|---|---|
| **$Op^5$** | **$Rd^3$** | **$Imm^8$** |

### *J-type format :*

5-bit opcode (Op) and11-bit Immediate

| | |
|---|---|
| **$Op^5$** | **$Imm^{11}$** |

### *Register Use*

For R-type instructions, Rs and Rt specify the source registers, and Rd specifies the destination register number. The function field F can specify at most four functions with the same opcode. For I-type instructions. The immediate constant is signed (range is -16 to +15), except for shift and rotate instructions (range is 0 to 31).

The B-type format is used by branch instructions, where Rd is a source register and the 8-bit signed immediate is used for PC-relative addressing. The J-type format is used by J (jump),

JAL (jump-and-link), and for constant formation (SET and SSET). The 11-bit immediate is used for PC-relative addressing and constant formation.

## ***Instruction Encoding***

The instructions, their meaning, and encoding are shown in the following table

| | ***Instruction*** | ***Meaning*** | ***Encoding*** | | | | |
|---|---|---|---|---|---|---|---|
| ***R- Type*** | *AND* | *Reg(Rd)= REg(Rs) & Reg(Rt)* | *OP=0* | *Rd* | *Rs* | *Rt* | *F=0* |
| | *OR* | *Reg(Rd)= REg(Rs) \| Reg(Rt)* | *OP=0* | *Rd* | *Rs* | *Rt* | *F=1* |
| | *XOR* | *Reg(Rd)= REg(Rs) ^ Reg(Rt)* | *OP=0* | *Rd* | *Rs* | *Rt* | *F=2* |
| | *EQV* | *Reg(Rd)= ~ REg(Rs) ^ Reg(Rt)* | *OP=0* | *Rd* | *Rs* | *Rt* | *F=3* |
| | *Add* | *Reg(Rd)= REg(Rs) + Reg(Rt)* | *OP=1* | *Rd* | *Rs* | *Rt* | *F=0* |
| | *Sub* | *Reg(Rd)= REg(Rs) - Reg(Rt)* | *OP=1* | *Rd* | *Rs* | *Rt* | *F=1* |
| | *SLt* | *Reg(Rd)= (REg(Rs)* <s *Reg(Rt))* ?1:0 | *OP=1* | *Rd* | *Rs* | *Rt* | *F=2* |
| | *SEQ* | *Reg(Rd)* = ( *REg(Rs) == Reg(Rt)*)? 1:0 | *OP=1* | *Rd* | *Rs* | *Rt* | *F=3* |
| ***I -Type*** | *AndI* | *Reg(Rd) = REg(Rs)* & (Imm5) | *4* | *Rd* | *Rs* | Imm5 | |
| | *ORI* | *Reg(Rd)= REg(Rs)* \| (Imm5) | *5* | *Rd* | *Rs* | Imm5 | |
| | *XORI* | *Reg(Rd)= REg(Rs)* ^ (Imm5) | *6* | *Rd* | *Rs* | Imm5 | |
| | *EQVI* | *Reg(Rd)* = ~ *REg(Rs)* ^ (Imm5) | *7* | *Rd* | *Rs* | Imm5 | |
| | *AddI* | *Reg(Rd)= REg(Rs)* + signed (Imm5) | *8* | *Rd* | *Rs* | Imm5 | |
| | *SLTI* | *Reg(Rd)* = ( *REg(Rs)* **<s** signed(Imm5))?1:0 | *9* | *Rd* | *Rs* | Imm5 | |
| | *SEQI* | *Reg(Rd)* = ( *REg(Rs)* == signed(Imm5))?1:0 | *10* | *Rd* | *Rs* | Imm5 | |
| | *SLL* | *Reg(Rd) = REg(Rs)* << unsigned(Imm5) | *11* | *Rd* | *Rs* | Imm5 | |
| | *SRL* | *Reg(Rd)= REg(Rs)* >> unsigned(Imm5) | *12* | *Rd* | *Rs* | Imm5 | |
| | *ROR* | Rotate Right *REg(Rs)* | *13* | *Rd* | *Rs* | Imm5 | |
| | *BEQ* | Branch if *REg(Rs) == Reg(Rd)* | *14* | *Rd* | *Rs* | Imm5 | |
| | *BNE* | Branch if *REg(Rs)* != *Reg(Rd)* | *15* | *Rd* | *Rs* | Imm5 | |
| | *LW* | *Reg(Rd)* = MEM[ *REg(Rs)* + signed(Imm5)] | *16* | *Rd* | *Rs* | Imm5 | |
| | SW | MEM[ *REg(Rs)* + signed(Imm5)]= *Reg(Rd)* | 17 | Rd | Rs | Imm5 | |
| ***B-Type*** | BEQZ | Branch if ( *Reg(Rd)* == 0) | *20* | *Rd* | *Imm8* | | |
| | BNEZ | Branch if ( *Reg(Rd)*!= 0) | *21* | *Rd* | *Imm8* | | |
| | BLTZ | Branch if ( *Reg(Rd)* < 0) | *22* | *Rd* | *Imm8* | | |
| | BGEZ | Branch if ( *Reg(Rd)*>= 0) | *23* | *Rd* | *Imm8* | | |
| | BGTZ | Branch if ( *Reg(Rd)* > 0) | *24* | *Rd* | *Imm8* | | |
| | BLEZ | Branch if ( *Reg(Rd)* <= 0) | *25* | *Rd* | *Imm8* | | |
| | JR | PC = *Reg(Rd)* + signed(Imm8 << 1) | *26* | *Rd* | *Imm8* | | |
| | JALR | R7=PC+2; PC = *Reg(Rd)* + signed(Imm8<<1) | *27* | *Rd* | *Imm8* | | |
| | SET | *Reg(Rd)* = signed(Imm8) | *28* | *Rd* | *Imm8* | | |
| | SSET | *Reg(Rd)*={ *Reg(Rd)*[23:0], Imm8} | *29* | *Rd* | *Imm8* | | |

| | | | | |
|---|---|---|---|---|
| ***J-Type*** | J | PC = PC + signed(Imm11<<1) | *30* | *Imm11* |
| | JAL | Reg(R7) = PC + 2;<br>PC= PC + signed(Imm11<<1) | *31* | *Imm11* |

## ***Instruction Description***

Opcodes 0 and 1 are used for R-type ALU instructions. Opcodes 4 through 13 are used for I-type ALU instructions. Register *Rd* is the destination register. The I-type ALU instructions (ANDI through SEQI) have identical functionality as their corresponding R-type instructions (AND through SEQ), except that the second ALU operand is immediate (Imm5). The 5-bit immediate constant is zero-extended for all Logical instructions ( with range 0 to 31) , and sign-extended for all other I-type instructions ( With range -16 to +15). The shift and rotate instructions use the lower 5 bits of Immediate constant as the shift/rotate amount with values 0 to 31

Opcodes 16 and 17 define the load word (LW) and store word (SW) instructions. These two Instructions address 32-bit words in memory. Displacement addressing is used. The effective memory address = Reg(Rs)+ sign_extend(Imm5). Register Rd is a destination register for LW, but a source for SW. Loading/storing a byte or half word are not defined to simplify the project. The I-type format is also used by BEQ, BNE. For the BEQ and BNE instructions, the 5-bit immediate specifies the branch target offset.

Opcodes 20 through 25 define six B- type branch instructions. The 32-bit value of reg(Rd) is read and compared against zero. PC-relative addressing is used to define the target of a branch instruction. If the branch is taken, the 8-bit immediate (Imm8) is left-shifted 1 bit and sign-extended then added to PC as follows:

If (branch is taken) PC = PC + sign_extend (Imm8<<1) else PC = PC + 2.

The PC register stores the address of a 16-bit instruction memory. The address is always multiple of 2 (least-significant bit of PC register is always 0).

The JR (Jump-Register) instruction does a register-indirect jump, where Imm8 is left-shifted 1 bit and sign-extended: PC = Reg(Rd) + sign_extend(Imm8<<1). The JALR (Jump-And-Link-Register) instruction saves the return address (PC+2) in R7.

The SET instruction (opcode 28) sets destination register Rd with an 8-bit signed constant. The immediate constant is sign-extended to 32 bits before writing in register Rd. The SSET instruction (opcode 29) reads and writes register Rd. It shifts the value of register a left 8 bits and sets the lower 8 bits: Rd = {Rd[23:0], Imm8}, where {} means concatenation. The SET and SSET instructions can be used together to form any 32-bit constant. For example, to initialize register R1 with constant 0x12345678, do the following:

SET R1, 0x12 (first byte)
SSET R1, 0x34 (second byte)
SSET R1, 0x56 (third byte)
SSET R1, 0x78 (fourth byte)

Opcodes 30 and 31 define the jump (J) and jump-and-link (JAL) instructions. PC-relative addressing is used to compute the jump target address where (Imm11) is left-shifted 1 bit and sign-extended:

PC = PC + sign_extend(Imm11<<1).

In addition, the JAL instruction writes the return address (PC + 2) in register R7.

## ***Memory***

Your processor will have separate instruction and data memories. The PC register should be restricted to 20 bits. The instruction memory can store $2^{19}$ instructions, where each instruction occupies two bytes. The data memory will be also restricted to $2^{20}$ bytes ($2^{18}$ words, each word is 32 bits or 4 bytes). Words should be always aligned in memory. The least-significant two bits of the data address must be zeros,

## Register File

Implement a Register file containing eight 32-bit registers R0 to R7 with two read ports and one write port.R0 is a normal register that can be read and written (NOT hardwired to zero)

## Arithmetic and Logical Unit (ALU)

Implement a 32-bit ALU to perform all the required operations (shown in tables above)

## Program Execution

The program will be loaded and will start at address 0 in the instruction memory. The data segment will be loaded and will start also at address 0 in the data memory. You may also have a stack segment if you want to support procedures. The stack segment can occupy the upper part of the data memory and can grow backwards towards lower addresses. The stack segment can be implemented completely in software. You can dedicate register R6 as the stack pointer. To terminate the execution of a program, the last instruction in the program can jump or branch to itself indefinitely (because there is no underlying operating system to terminate the program).

## Phase 1: Building a Single cycle Processor

It is recommended that you start by building the datapath and control of a single-cycle processor and ensure its correctness. You should have sufficient test cases that ensure the correct execution of ALL instructions in the instruction set. You should also write test cases that show the correct execution of complete programs. To verify the correctness of your design, show the values of all registers (R0 to R7) at the top-level of your design. Provide output pins for registers R0 through R7, and make their values visible at the top level of your design to simplify testing and verification.

## Phase 2: Building Pipeline processor

Once you have succeeded in building a single-cycle processor and verified its correctness, design and implement a pipelined version of your design. Make a copy of your single-cycle design, then convert it and implement a pipelined datapath and its control logic. Add pipeline registers between stages. Design the control logic to detect data dependencies among instructions and implement the forwarding logic. You should handle properly the control hazards of the branch and jump instructions. Also, stall the pipeline after a LW instruction, if it is followed by a dependent instruction.

## *Testing and verification*

To demonstrate that your CPU is working, you should do the following:

- Test all components and sub-circuits independently to ensure their correctness. For example, test the correctness of the ALU, the register file, the control logic separately, before putting your components together.
- Test each instruction independently to ensure its correct execution.
- Write a sequence of instructions to verify the correctness of ALL instructions. Use SET and SSET to initialize registers or load their values from memory. Demonstrate the correctness of all ALU R-type and I-type instructions. Demonstrate the correctness of LW and SW instructions. Similarly, you should demonstrate the correctness of all branch and jump instructions
- Test sequences of dependent instructions to ensure the correctness of the forwarding logic. Also, test a LW (load word) followed by a dependent instruction to ensure stalling the pipeline correctly by one clock cycle.
- Test the behavior of taken and untaken branch instructions and their effect on stalling the pipeline.
- Make several copies and versions of your design before making changes, in case you need to go back to an older version.

## *Test Programs:*

Write a test program and translate it into machine instructions. Load the instructions into the instruction memory starting at address 0. Also save the image of the instruction and data memories into files and reload them later for testing purposes.

## ***Project Report***

The report document must contain sections highlighting the following:

### *1. Design and Implementation*

- Specify clearly the design giving detailed description of the datapath, its components, control, and the implementation details.
- Provide drawings of the component circuits and the overall datapath.
- Provide a description of the control logic and the control signals. **Provide a table giving the control signal values for each instruction. Provide the logic equations for each control signal.**
- Provide a description of the forwarding logic, the cases that were handled, and the cases that stall the pipeline, and the logic that you have implemented to stall the pipeline

### *2. Simulation and Testing*

- Carry out the simulation of the processor developed using Logisim.
- Describe the test programs that you used to test your design with enough comments describing the program, its inputs, and its expected output. List all the instructions that were tested and work correctly. ***List all the instructions that do not run properly.***
- Document all your test programs and files and include them in the report document
- Describe all the cases that you handled involving dependences between instructions, forwarding cases, and cases that stall the pipeline
- Also provide snapshots of the Simulator window with your test program loaded and showing the simulation output results.

## ***Submission Guidelines***

- ❖ The single-cycle processor design should be completed at **28-5-2021** .
  - ➢ It should be fully operational. You should have sufficient test cases ready to prove that your CPU is fully functional.
- ❖ The pipelined processor design should be completed at **4-6-2021** .
  - ➢ It should be fully operational. You should have sufficient test cases ready to prove that your pipelined CPU is fully functional.
- ❖ If your CPU is not fully operational then identify which instructions do not work properly, or which hazards are not handled properly to avoid the loss of many marks.

❖ All submission will be done through both emails:

- ➢ gaa11@fayoum.edu.eg
- ➢ gna00@Fayoum.edu.eg

❖ The project should be submitted **on the due date by midnight.**

❖ Attach one zip file containing:

- All the design circuits and sub-circuits,
- The test programs, their source code and binary instruction files that you have used to test your design, their test data, as well as the report document.
- Project report
- ***Video demonstrating your work in details.***

## Grading policy

The grade will be divided according to the following components:

- **Correctness: whether your implementation is working**
- **Completeness and testing: whether all instructions and cases have been implemented, handled, and tested properly**
- **Report document**