Mini RISC Processor

Introduction

- A 16 bit RISC Processor
- 4 stage pipeline
- Fixed instruction length
- 4kB Instruction Memory, 4kB Data Memory

Tools Used

- **Hardware/Simulation Tools**: Xilinx Vivado, ModelSim,
  FPGA: Basys 3 Artix 7
  Software: Verilog for RTL design.
- **Languages**: Verilog for design
- **Testing Tools**: Testbenches, simulation scripts.

Addressing Modes

- Register to Register Addressing (R - Type)

  | Opcode | Reg1 | Reg2 |Res |

  | 5 - bit |3 - bit | 3-bit | 3 - bit |

- Immediate Addressing

  | Opcode | Dest_reg| Immediate_Val |

  | 5 - bit | 3 - bit | 8 - bit |

Instruction Set

We have 7 types of Instructions: (refer to the excel sheet for detailed syntax

- **Special Instructions** `(000xx)`
  - NOP, HALT, CALL, RET

- **ALU Instructions** `(001xx)`
  - ADD, SUB, MUL, DIV

- **Logic Instructions** `(010xx)`

    - NOT, AND, OR, XOR

- **LOAD/ STORE Instructions** `(011xx)`

    - LUB (Load Upper Byte) : Loads Immediate Byte into a register (R0 - R7).
      `LUB R2, #04`

    - LLB (Load Lower Byte) : Loads Immediate Byte into a register (R0 - R7).
      `LLB R1, #04`

    - LOAD : Load the value stored at the address held in a register into another register.
      `LOAD [R_src], R_dest`

    - STORE: Stores the 16-bit value of a register to the memory location pointed to by another register

      `STORE_ADDR R_dest, [R_src]` ; *Store the value in R1 at the memory address held in R2*

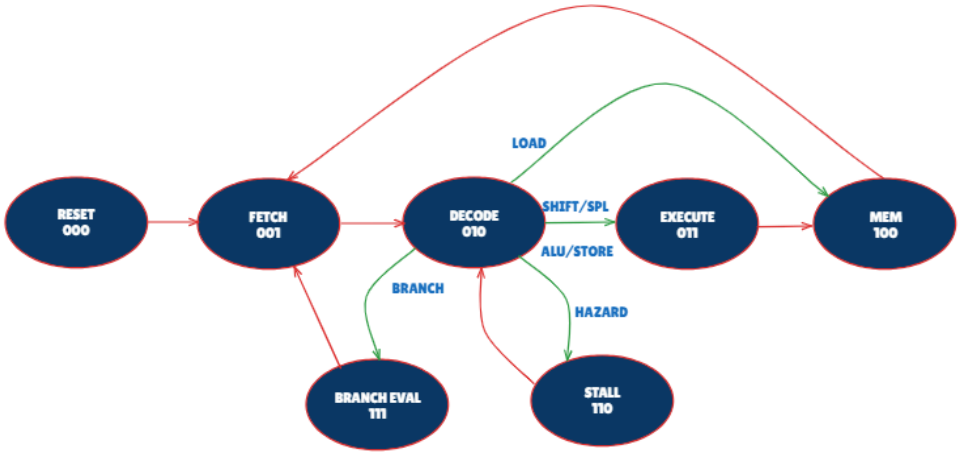
- **Branch Instructions** `100xx`
    - JMP, BEZ, BEQ, BNQ ---- (Conditional and Unconditional Jumps)

- **Extended ALU Instructions** `101xx`
    - CMP, RR, RL, INC
    - CMP: Compares two registers and sets the cmp flag and eq flag.
        - `CMP R1, R2` *; if R1 > R2 => cmp is set, eq is cleared*
          ; if R1 < R2 => cmp is cleared, eq is cleared
          ; if R1 = R2 => eq is set, cmp is cleared

- **Stack Instructions** `110xx`
    - PUSH, POP

- **Extra Branch Instructions** `111xx`
    - BNF, BLT, BGT

The Control Unit FSM

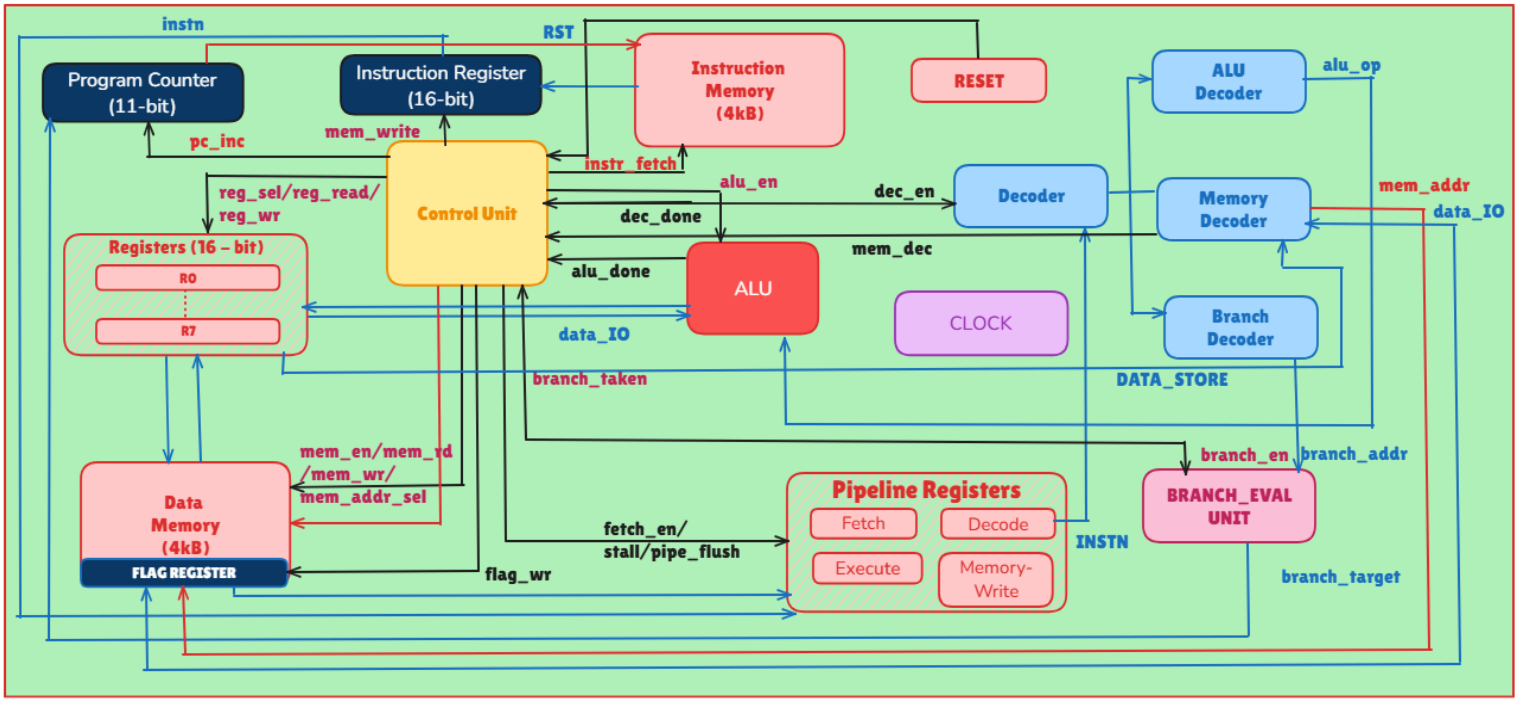


States in the FSM

FSM State	Control Signals	Purpose
RESET (000)	`reset`	Initializes the pipeline, sets the PC to 0, clears registers and memory initialization.
FETCH (001)	`pc_inc`, `instr_fetch`	Fetches the instruction from memory using the PC. Increments the PC to point to the next instruction.
DECODE (010)	`dec_en`, `alu_en`, `branch_en`, `mem_en`	Activates appropriate decoders (main decoder and sub-decoders like ALU, Branch, or Memory).
	`dec_done`	Indicates that decoding is completed and the instruction type is identified.
EXECUTE (011)	- **ALU Operations:** `alu_op`, `alu_done`	Activates ALU operations (e.g., ADD, SUB, MUL) and signals operation completion.
	- **Branch Ops:** `branch_eval`, `branch_taken`	Evaluates branch conditions and updates PC if branch is taken.

	- **Memory Ops:** `mem_addr`, `mem_rd`, `mem_wr`	Initiates memory read/write operations.
MEM (100)	`mem_en`, `mem_addr`, `mem_rd`, `mem_wr`	Accesses Data Memory to either read or write data.
	`reg_wr`	Writes back data to the destination register from memory (if applicable).
BRANCH EVAL (111)	`branch_target`, `branch_taken`, `pc_update`	Evaluates branch conditions, flushes pipeline if branch is taken, and updates PC accordingly.
STALL (110)	`stall`, `pipe_flush`	Halts pipeline stages temporarily to resolve hazards (e.g., data hazards, structural hazards).

1. **Global Signals**:
    - `pc_inc` increments the Program Counter for instruction fetch.
    - `pipe_flush` clears pipeline registers during branch mispredictions or hazard resolution.
2. **Decoder Control**:
    - `dec_en` enables the main decoder, which activates the relevant sub-decoders based on the instruction type.
    - Sub-decoders (`alu_en`, `branch_en`, `mem_en`) operate independently once activated.
3. **Stage-Specific Operations**:
    - Execute stage signals (`alu_op`, `branch_en`, etc.) are determined by sub-decoder outputs, reducing FSM complexity.



Memory Organization

DATA MEMORY

We have 4kB of Data memory.
Using Word addressing. So, 11 bit address bus.

SPECIAL FUNCTION REGISTERS

1. The I/O pins: We have 16 I/O pins, mapped to the address range 0xFF0 to 0xFFF
2. The Flag Register: (using bit-addressing) : 0xFFE
    a. Carry Flag (C)
    b. Overflow Flag (O)
    c. CMP Flag (cmp)
    d. Equal Flag (eq)
    e. General Purpose Flag (F)
    f. Parity Flag (P)
    g. Negative Flag (N)
    h. Zero Flag (Z)
3. The Stack: 128 bytes
4. ….

Address Range	Purpose
`0x000-0xFCF`	General Purpose Registers (
`0xFD0-0xFEE`	Stack (30 words / 60 Bytes / 480 bits) `0xFD0: Stack Pointer`
`0xFEF`	Flags (8 bits)
`0xFFF`	I/O Pins (16 bits)

Stack Memory Overflow

1. Software check
   ex:

```
POP R1          ; Increment SP
CMP SP, 0xFE7   ; Compare SP with stack end
BGT underflow   ; Branch to underflow handler if SP > 0xFE7
LOAD R1, [SP]   ; Load R1 from SP
JMP continue    ; Continue execution
underflow:
HALT            ; Handle underflow (or jump to error routine)
```

Branch Evaluation Unit Design (for Pipelining)

- Compute the target address for branch instructions, resolve the branching condition, and update the Program Counter (PC) accordingly.

- Relative addressing scheme.

Inputs to the BEU

1. **branch_en (Enable Signal):** A control signal from the Control Unit, which enables the BEU to evaluate the branch instruction. This signal ensures the BEU is only active during a branch instruction.

2. **branch_addr (Branch address):** The instruction fetched by the pipeline and decoded by the Branch Decoder. This includes:

   - The opcode specifying the type of branch (e.g., JMP, BEZ, BEQ, BNQ, BSF).
   - The relative address field (11-bit in this design), which specifies the offset from the current Program Counter value.

3. **flag_reg (Flag Register):** Contains condition flags (e.g., Zero, Equal) that determine whether conditional branches (BEZ, BEQ, BNQ, BS, BLT, BGT) are taken.

4. **pc_current (Current Program Counter):** The current address from the Program Counter, used as the base for calculating the target address.

Outputs of the BEU

1. **branch_target (Target Address):** The computed address to which the Program Counter should jump if the branch is taken.
2. **branch_taken:** A signal to the Control Unit indicating whether the branch condition has been met and the jump has been performed.

Internal Logic of the BEU

1. **Relative Address Calculation:**

    - The BEU adds the 11-bit relative address from the branch instruction to the current Program Counter (pc_current) to compute the target address.
    - Since relative addressing is used, the relative address can be positive or negative, represented in 2's complement form. The BEU handles this by sign-extending the 11-bit offset to match the bit-width of the Program Counter (e.g., 16 bits).

    Formula:

    branch_target = pc_current +reg(branch_addr[10:0])

2. **Condition Evaluation:**

    - The BEU evaluates the branch condition based on the opcode and the flag register.
        - **Unconditional Branch (JMP):** Always taken; no flags are checked.
        - **Branch Equal to Zero (BEZ):** Taken if the Zero flag is set.
        - **Branch Equal (BEQ):** Taken if the Equal flag is set.
        - **Branch Not Equal (BNQ):** Taken if the Equal flag is not set.
        - **Branch if Flag is set (BSF)**: Taken if the General Purpose flag is set.
        - **Branch Less Than (BLT)** : Taken if the CMP Flag is cleared.
        - **Branch if Greater Than (BGT) :**Taken if the CMP Flag is set.
    - If the condition is satisfied, `branch_taken` is asserted, and the computed `branch_target` is sent to the Program Counter.
    - If the condition is not satisfied, `branch_taken` is de-asserted, and the Program Counter continues incrementing sequentially.

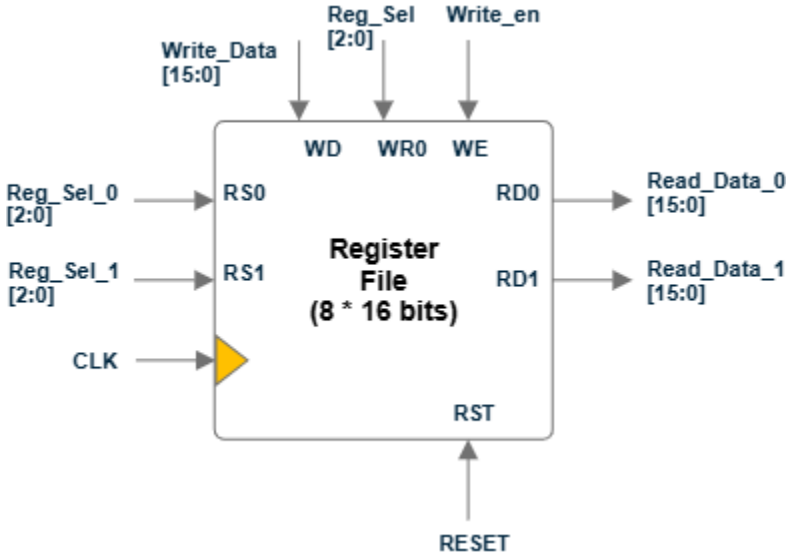
3. **Integration with Control Unit:**

    - The BEU sends the `branch_taken` signal to the Control Unit to synchronize the update of the Program Counter.
    - The Control Unit manages pipeline control signals like `fetch_en`, `stall`, and `pipe_flush` based on the branch decision.

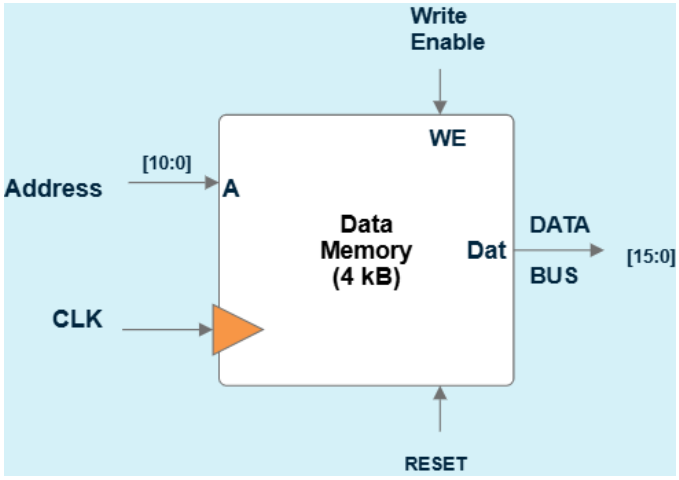
Additions, as discussed in Meeting, 06/Jan

Design of each module

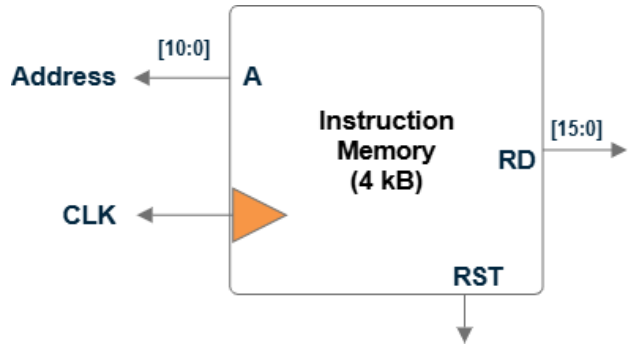
Register File



Data Memory

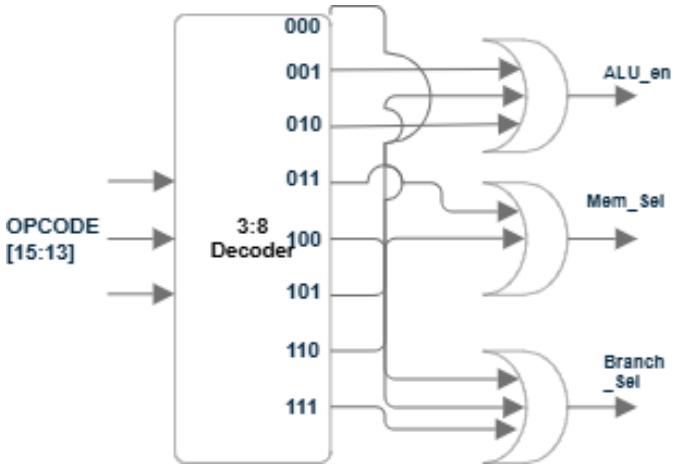


Instruction Memory

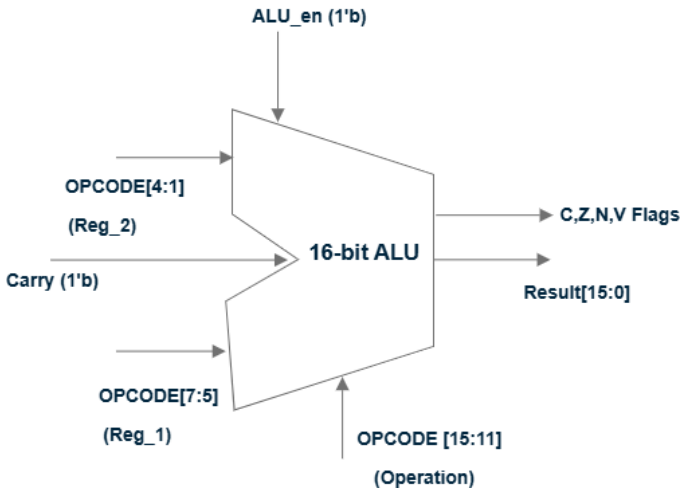


Decoder Unit (for non-pipelined processor)

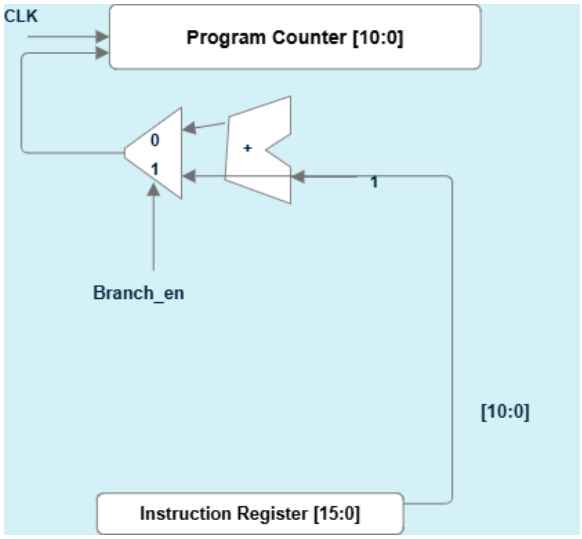
Main Decoder



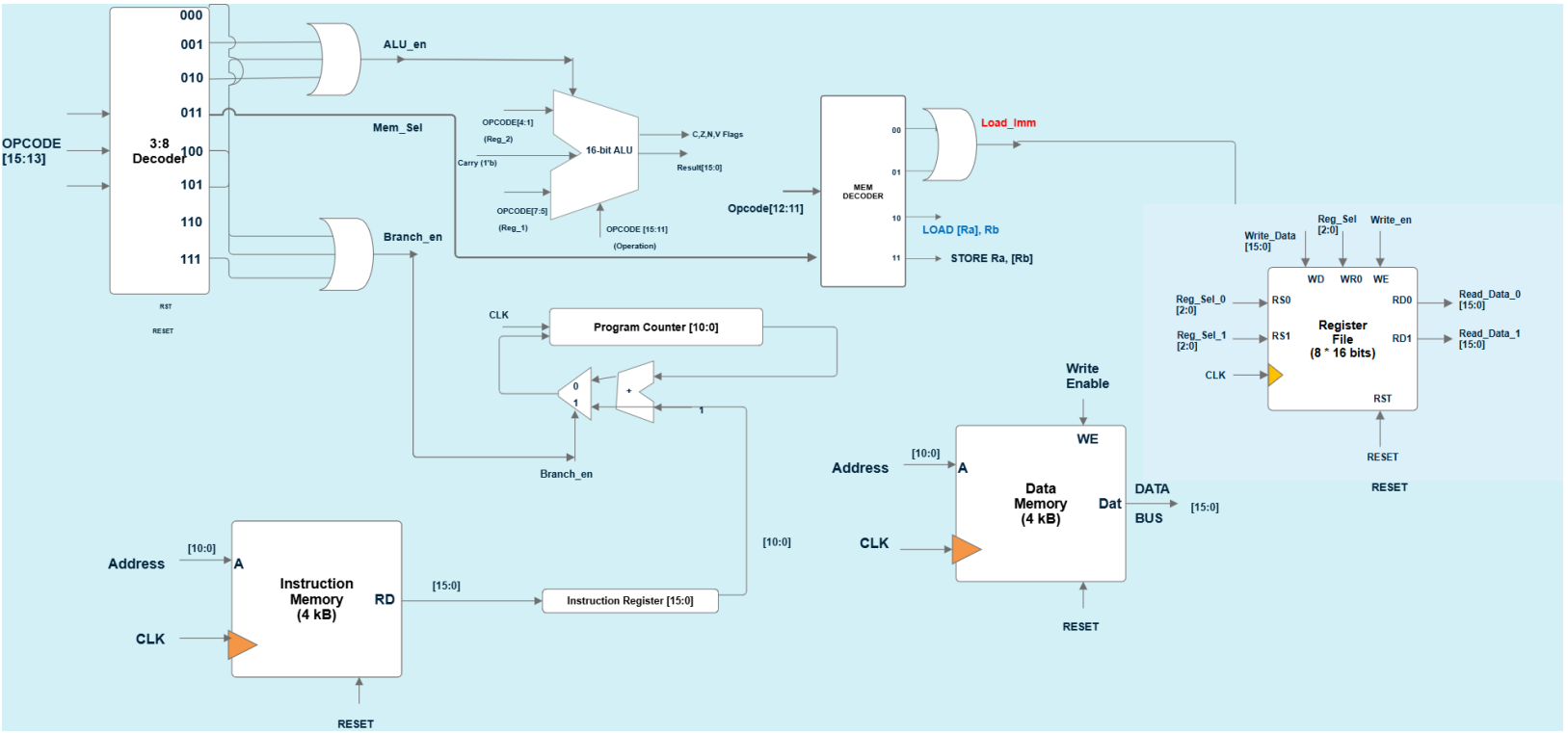
ALU



Program Counter



Integrated Diagram



Organize the Instruction and Data Flow model

A program:

LLB R4, #0D

LOAD R2, [R4]

LLB R4, #0E

LOAD R3, [R4]

ADD R1, R2, R3 ;which adds R3 and R2 and puts the result in R1

LOAD R4, #0F

STORE [R4], R1

When I have to load an immediate value in the first instruction, the mem decoder unit gives the Load_Imm signal that connects to the write_en of the Register file. The relevant bits of the opcode are sent to the reg_sel and the immediate value in the data bus of the register file.

Then, since it is a load instruction, the value in R4 is put on the address bus of the register file, which connects to the address bus of the Data memory. Then the contents in the memory location move to the data bus of the data memory and that enters the register specified in the instruction.

Now, we have to add. The ALU receives the enable signal, based on the first 3 bits of the opcode. Then, all five opcode bits are supplied to it, to select the operation. It then has to add the contents in the registers and put it to the specified result register, R1, in this case. (Here's where I'm confused, how will the ALU actually do this? I mean I intend to write Verilog behavioural logic code, but I'd like to know how the ALU accesses these).

Then finally, the contents are stored into the Data memory.

Work Plan

Phase 1: Planning and Component Design for Non-pipelined Processor (Weeks 1-4: Jan 6th to Feb 1st)

- **Week 1 and Week 2:**
    - Finalize the instruction set architecture (ISA) define the number of registers, addressing modes, and all instructions
    - Begin designing the datapath modules
    - Finish designing the datapath modules and start writing their Verilog code.
- **Week 3:** Complete the Verilog coding, simulation, debugging using testbenches. Integrate the modules.
- **Week 4:** Implement the non-pipelined processor on the Basys 3 FPGA.

Phase 2: Pipelined Processor Development (Weeks 5-8: Feb 3rd to Mar 1st)

- **Week 5 and 6:** Research, Analyze, Design the pipeline architecture of the processor. Address potential pipeline hazards (data, control, structural).
- **Week 7 and 8:** Complete Verilog coding, write a testbench.

Phase 3: Pipeline Implementation (Weeks 9-12: Mar 3rd to Mar 29th)

- **Week 9 and 10:** Implement the pipelined processor on the FPGA.
- **Week 11 and 12:** Buffer time.

Phase 4: Documentation and Presentation (Week 13: Mar 31st to Apr 4th)

- **Week 13:** Presentation.