Design of a 16-bit CPU (datapath & controller)

NOTE: Please read the whole lab sheet before attempting to work on the solution.

OBJECTIVE

To design a 16-bit CPU by integrating smaller building blocks (i.e., ALU, DMEM, IMEM, RF sub-modules) and an instruction decoder as a complete CPU datapath. Then to integrate the datapath with the corresponding controller of the specified instruction set. You will build the sub-modules first, then integrate them into a working CPU.

Let us call our 16-bit processor “ADSD-RISC”.

RESOURCES

Quartus II, DEO-Nano board

PART 1: Top Level

1.1 CPU Microarchitecture Specifications

The CPU is split into a datapath and a controller. The overall datapath for the 16-bit ADSD-RISC is shown in Figure 1.1, consisting of several modules including a 16-bit ALU, byte-addressable instruction memory (IMEM), byte-addressable data memory (DMEM), 16-bit register file (RF), and other required components such as the program counter (PC), 16-bit sign_ext logic, and a few multiplexers. The control signals for the multiplexers (as well as PC and IR, if any) are not shown in the diagram. The CPU will execute each instruction in one cycle. No pipeline.

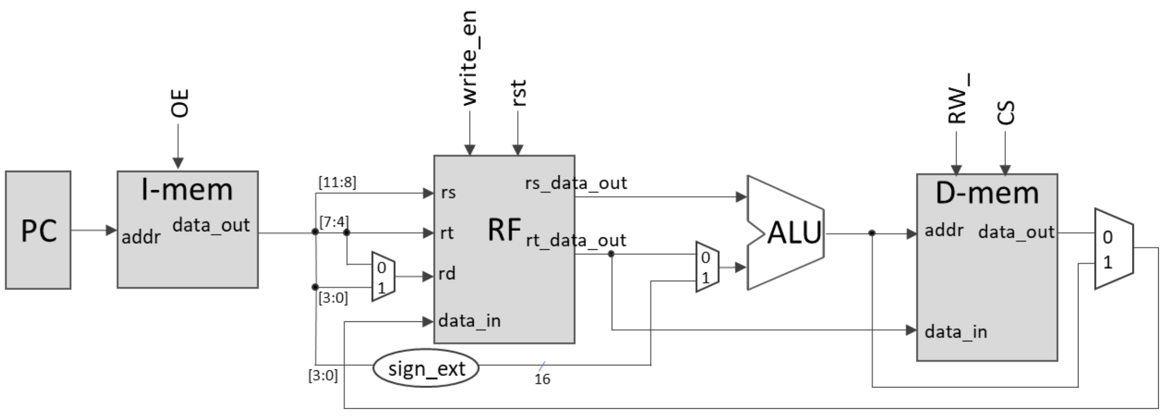


Figure 1.1 Simplified block diagram of the single cycle computer microarchitecture of ADSD-RISC

1.2 Instruction Format for 16-bit ADSD-RISC ISA

	15 14 13 12	11 10 9 8	7 6 5 4	3 2 1 0
R-type	opcode	rs	rt	rd
S-type	opcode	rs	rt	#imm4
B-type	opcode	rs	rt	#imm4
L-type	opcode	rs	rt	#imm4
J-type	opcode	#imm12		

R-type	: add, sub, or, and
S-type	: shl, shr, rol, ror, not
B-type	: beq, blt, bgt
L-type	: ld, st
J-type	: jmp

1.3 Program Counter (PC)

The PC register is used to store the next address of the instruction to be loaded and executed. It needs the capability to be incremented by the quantum required to access the next address in the IMEM, which depends on your IMEM design.

1.4 Instruction Register (IR)

The IR register (not shown in Fig 1.1) does not have to be an actual register to store the instruction that comes out of the IMEM. For this implementation, the different bits of the "supposed" IR need to be connected to the different input ports of the RF module. Simple wires would suffice. The choice of bits depends on the instruction decoding requirements for the ISA (instruction set architecture). The IR is therefore not illustrated in Fig 1.1.

1.5 Sign Extension (sign_ext)

This combinational circuit copies the sign bit of the incoming bits from the IR to make up the 16-bits required as the second input to the ALU.

1.6 Controller (CTRL)

A controller (Fig. 1.2) is required in order to control the ALU to perform the necessary action required by the instruction. Based on the instruction set (read ISA) of our ADSD-RISC CPU, the controller only needs the 4-bit opcode as input. For each opcode, it is supposed to provide the necessary control signals to enable the datapath to perform the tasks specified by the instruction currently being decoded and executed.

In addition, the PC incrementor (left most block in Fig. 12) is also required so that the subsequent instruction can be fetched at the next clock cycle, which is integrated with the branch and jump control signals.

The diagram below illustrates how the controller is interfaced to the datapath.

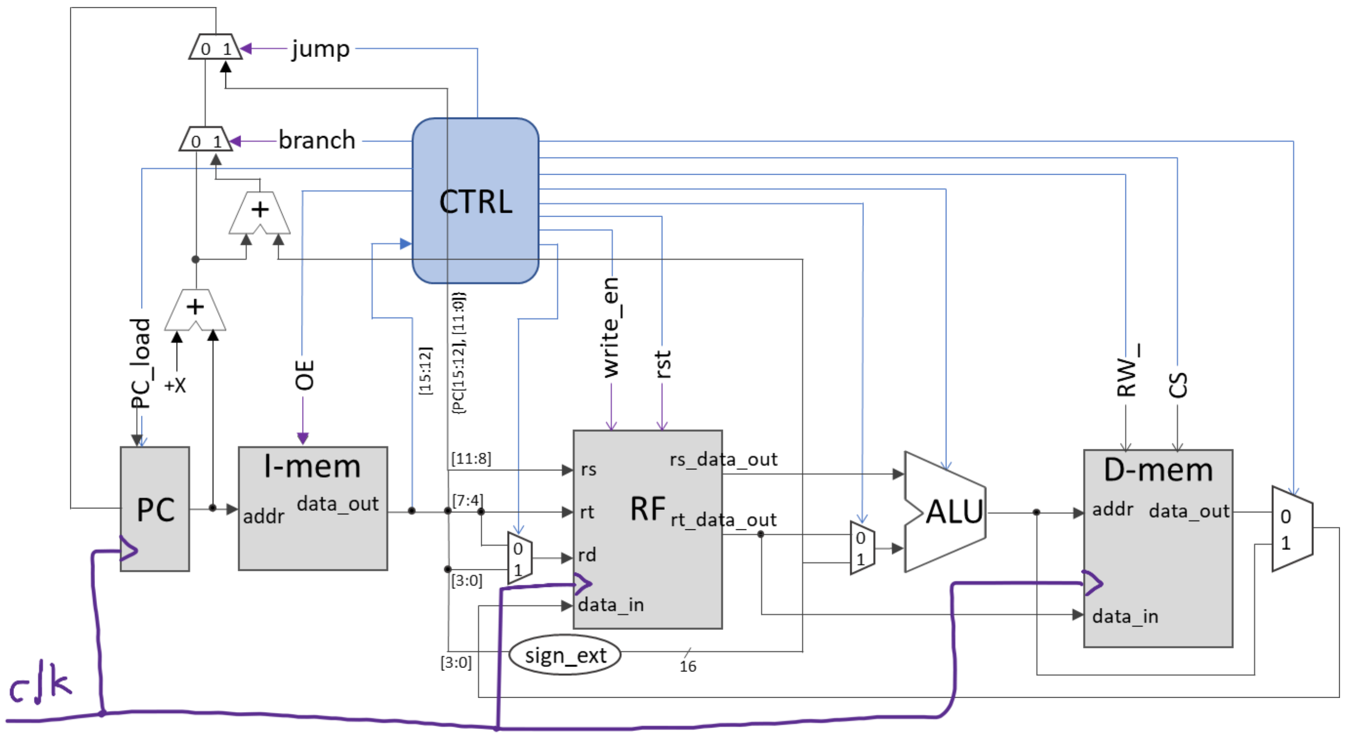


Figure 1.2 Block diagram of the single cycle microarchitecture of ADSD-RISC with the controller module.

PART 2: Subsystem Modules

2.1 Arithmetic Logic Unit (ALU)

An arithmetic logic unit (ALU) is a combinational digital electronic circuit that performs arithmetic and bitwise operations on signed integer binary numbers. This is in contrast to a floating-point unit (FPU), which operates on floating point numbers. An ALU is a fundamental building block of many types of computing circuits, including the central processing unit (CPU) of computers.

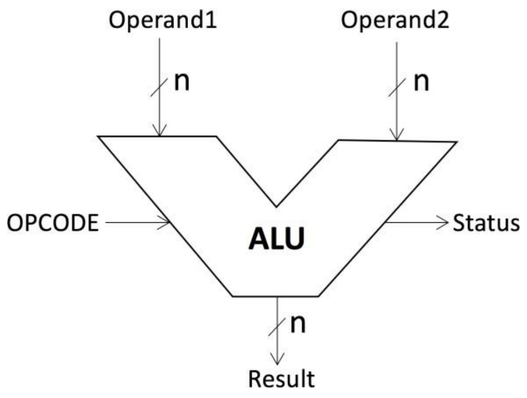


Figure 2.1 Arithmetic logic unit block diagram

The inputs to an ALU (Figure 2.1) are the data to be operated on, called operands, and a code (opcode) indicating the operation to be performed and, optionally, status information from a previous operation; the ALU's output is the result of the performed operation. In many designs, the ALU also exchanges additional information with a status register, which relates to the result of the current or previous operations. The status output is used by the controller to make certain decisions, especially for branch instructions, where applicable.

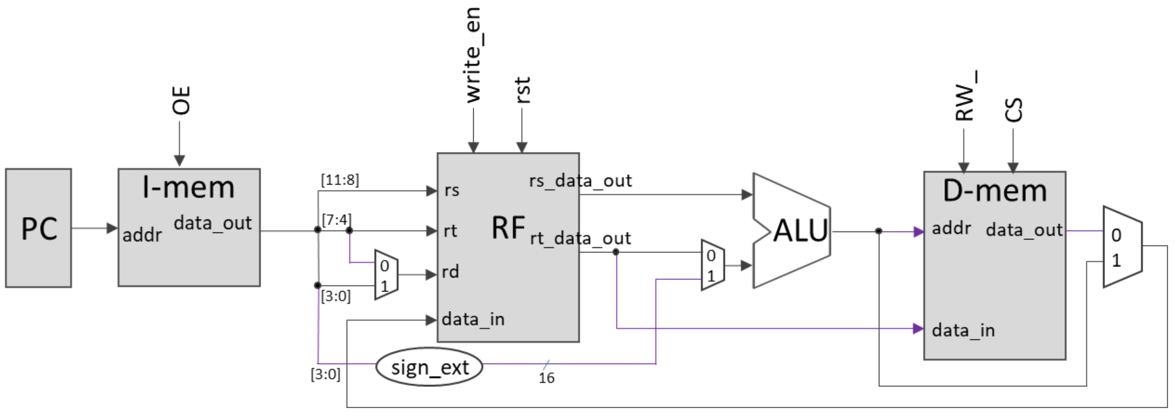
A number of basic arithmetic and bitwise logic functions are commonly supported by ALUs. Basic, general purpose ALUs typically includes these operations in their repertoires:

- Arithmetic operations
- Bitwise logical operations
- Bit shift operations
- Conditional branch operations
- Jump operations

In this lab, you have to design a 16-bit ALU with 13 instructions group into four different types. Details of each instruction are provided in Sections 2.1.1 – 2.1.5.

2.1.1 R-type : Register instructions

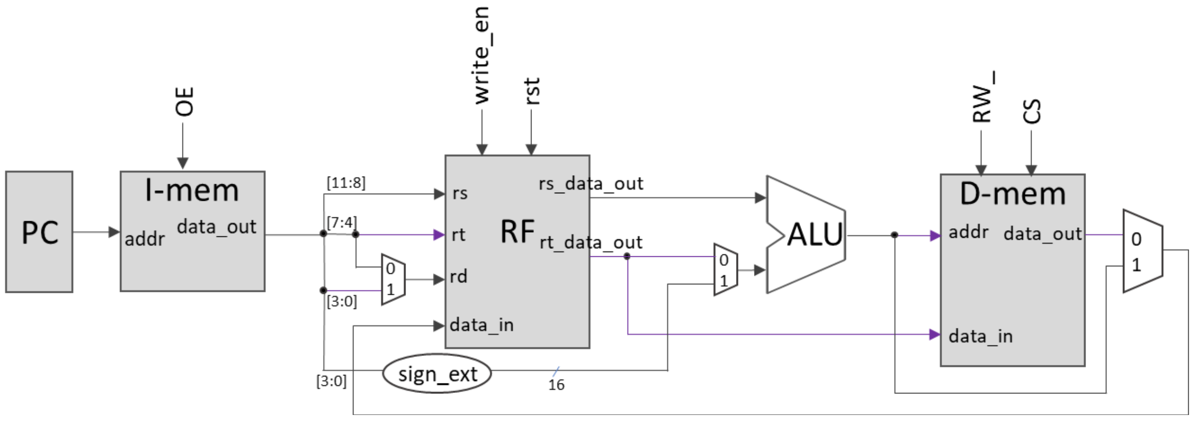
Opcode	Instruction	RTL Operation
0000	add rs, rt, rd	rd ← rs + rt
0001	sub rs, rt, rd	rd ← rs + rt
0010	or rs, rt, rd	rd ← rs | rt (bitwise OR)
0011	and rs, rt, rd	rd ← rs & rt (bitwise AND)



*Datapath used by R-type instructions are shown in black. Blue paths are unused.

2.1.2 S-type : Shift instructions

Opcode	Instruction	RTL Operation
0100	shl rs, rt, #imm4	rt ← arithmetic shift left rs by #imm4 bits
0101	shr rs, rt, #imm4	rt ← arithmetic shift right rs by #imm4 bits
0110	rol rs, rt	rt ← {rs[14:0], rs[15]}
0111	ror rs, rt	rt ← {rs[10], rs[15:1]}
1000	not rs, rt	rt ← ~rs //bitwise negation
1111	addi rs, rt, #imm4	rt ← rs + {12{imm4[3], imm4}}



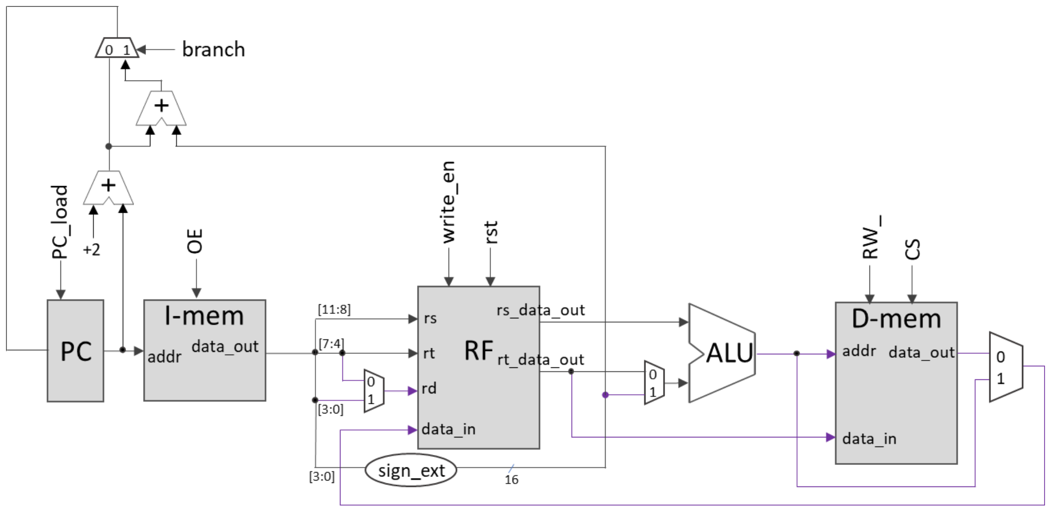
*Datapath used by S-type instructions are shown in black. Blue paths are unused.

2.1.3 B-type : Conditional branch instructions

Opcode	Instruction	RTL Operation
1001	beq rs, rt, #imm4	If rs == rt, PC ← PC + 2 + #imm4, else PC ← PC + 2
1010	blt rs, rt, #imm4	If rs < rt, PC ← PC + 2 + #imm4, else PC ← PC + 2
1011	bgt rs, rt, #imm4	If rs > rt, PC ← PC + 2 + #imm4, else PC ← PC + 2

* #imm4 is sign-extended to 16 bits

* depending on whether your IMEM is byte addressable or double byte addressable, the PC is incremented by either +2 or +1, respectively. Amend the RTL operation accordingly in your design.

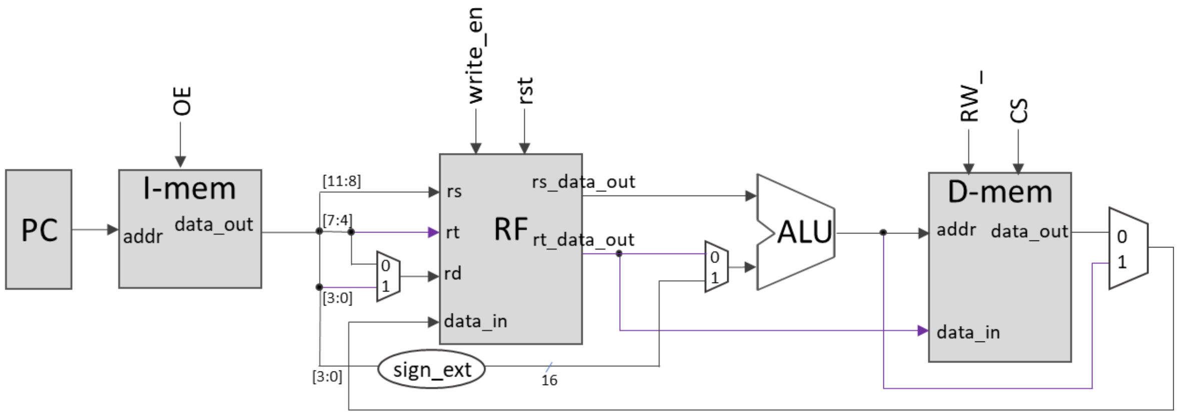


*Datapath used by B-type instructions are shown in black. Blue paths are unused.

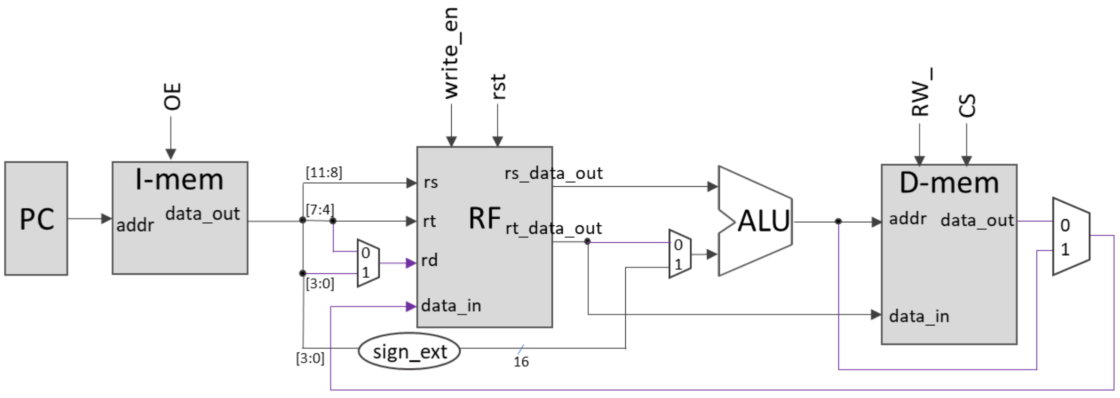
2.1.4 L-type : Load/store instructions

Opcode	Instruction	RTL Operation
1100	ld rs, rt, #imm4	rt ← DMEM [rs + #imm4]
1101	st rs, rt, #imm4	DMEM [rs + #imm4] ← rt

* #imm4 is sign-extended to 16 bits



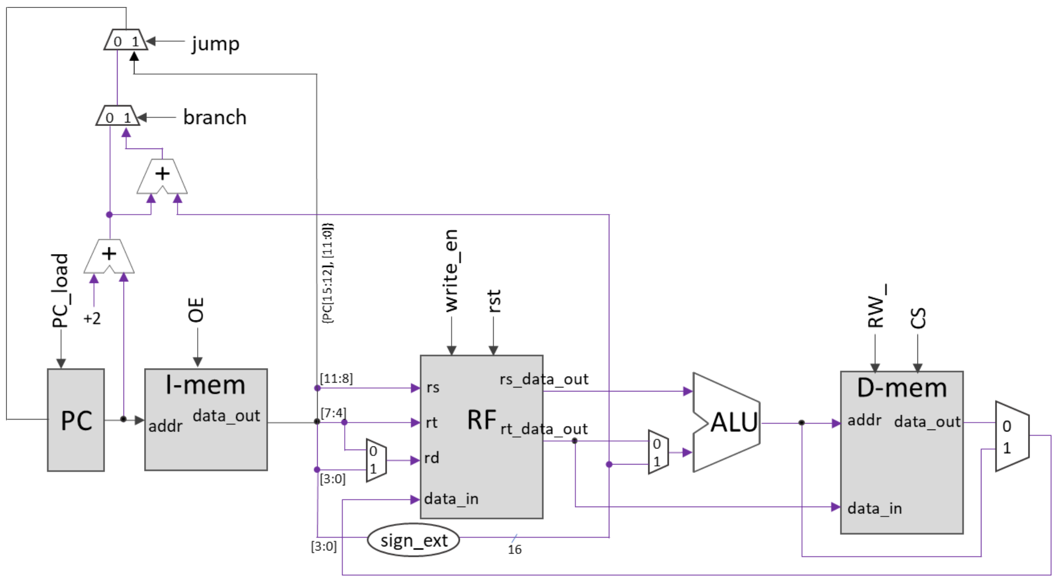
*Datapath used by load instruction is shown in black. Blue paths are unused.



*Datapath used by store instruction is shown in black. Blue paths are unused.

2.1.5 J-type : Jump instruction

Opcode	Instruction	Operation
1110	jmp #imm	PC ← {PC[15:12], #imm12}



*Datapath used by J-type instructions are shown in black. Blue paths are unused.

2.1.6 Other information on the ALU

The 16-bit ALU has the following inputs and outputs:

- A: 16-bit signed input
- B: 16-bit signed input
- Output: 16-bit signed output
- Output: status bits (zero, neg, ovf—see details below)
- Control: 4-bit control input

Status bits:
- zero bit is high when the arithmetic operation results in zero.
- neg bit is high when the arithmetic operation results in negative output.
- ovf bit is high when the arithmetic operation results in an overflow.

The following points should be taken care of:

- Use a case statement (or a similar 'combinational' statement) that checks the input combination of "opcode" and acts on the operands as described in Section 2.1.
- The above circuit for each instruction is completely combinational. The output should change as soon as the code combination or any of the input changes.
- You can use arithmetic and logical operators to realize your design.

2.1.7 Overflow Rule for Additions

If 2 Two's Complement numbers are added, and they both have the same sign (both positive or both negative), then overflow occurs if and only if the result has the opposite sign. Overflow never occurs when adding operands with different signs.

i.e.
- Adding two positive numbers must give a positive result.
- Adding two negative numbers must give a negative result.

Overflow occurs if;

- (+A) + (+B) = −C
- (−A) + (−B) = +C

Example: Using 4-bit Two's Complement numbers ($-8 \le x \le +7$)

```
 (-7)   1001
+(-6)   1010
------------
(-13) 1 0011 = 3 : Overflow (largest -ve number is -8)
```

2.1.8 Overflow Rule for Subtractions

If 2 Two's Complement numbers are subtracted (i.e. minuend minus subtrahend), and their signs are different, then overflow occurs if and only if the result has the same sign as the subtrahend.

Overflow occurs if;

- (+A) − (−B) = −C
- (−A) − (+B) = +C

Example: Using 4-bit Two's Complement numbers (−8 ≤ x ≤ +7)
Subtract −6 from +7

```
  (+7)  0111                 0111
 -(-6)  1010 -> Negate -> +0110
 ----------              -----
   13                     1101 = -8 + 5 = -3 : Overflow
```

2.2. Instruction Memory (IMEM) Specifications.

A computer system requires storage for its program and data to function properly. For this purpose, we need to design separate functional modules for ROM (read-only storage for program instructions) and RAM (read/write storage for data memory). Let us call the ROM as IMEM (instruction memory) and the RAM as DMEM (data memory).

1. Inputs:
   16-bit address (addr)
   1-bit output enable (OE).
   NOTE: When OE = 0, data_out is in high impedance (`16'bz`) state.

2. Outputs:
   16-bit data output (data_out)

3. Storage capacity:
   At least 1 kilobyte occupying the address space starting at 0x0000, sequentially.
   Note: You can decide whether to design the read-only memory array either as byte addressable where each address is 8 bits, but with 16-bit data output port or double-byte addressable where each address is 16-bits. If you choose the former, read access to any address *i* would return the 16-bit contents of two consecutive 8-bit memory addresses *i* and *i*+1, based on ~~little-endian~~ big-endian byte ordering like RISC-V. For this lab, you can program the ROM to contain random values for verification purposes.

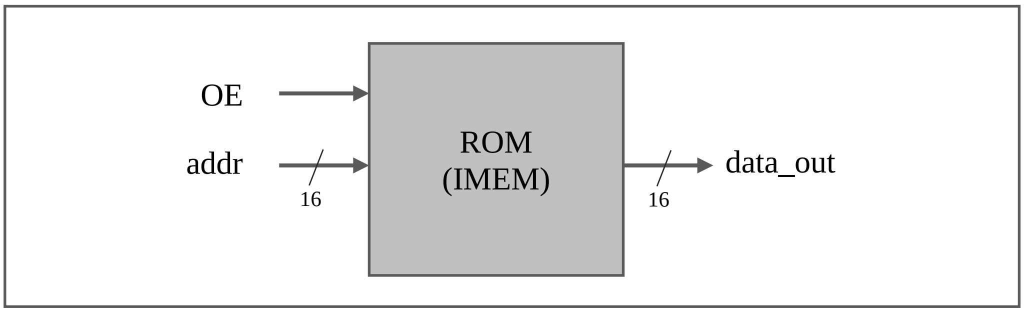


Figure 3.1 : IMEM block diagram

2.3 Data Memory (DMEM) Specifications

The data memory module is clocked. Read operation begins when the control signal RW_ = 1 (when CS=1), i.e. it behaves like a combinational circuit regardless of the clock edge/value. The write operation, on the other hand, is synchronized to the clock edge.

1. Inputs:
   1-bit clock (clk)
   16-bit address (addr)
   16-bit data input (data_in)
   1-bit R/W' (RW_) → i.e. RW_ = 1 (read), RW_ = 0 (write)
   1-bit CS (chip select). → i.e. RAM does not allow read write when CS = 0.
   When CS = 0, the RAM output (data_out) should be high impedance (`16'bz`) state.
2. Outputs:
   16-bit data output (data_out).
   Note: Output is zero if the read address is beyond the memory address space.
   If CS = 0, output should be in high impedance state or tri-state value.

3. Storage capacity:
   At least 1 kilobyte occupying the address space starting at 0x0000, sequentially.
   Note: The memory array should be designed as **byte addressable**, but with 16-bit data input and output port. Read access to any address *i* would return the 16-bit contents of two consecutive 8-bit memory addresses *i* and *i*+1, based on little-endian byte ordering like RISC-V, similarly for a write transaction. The memory does not have to be initialized to all zeros.

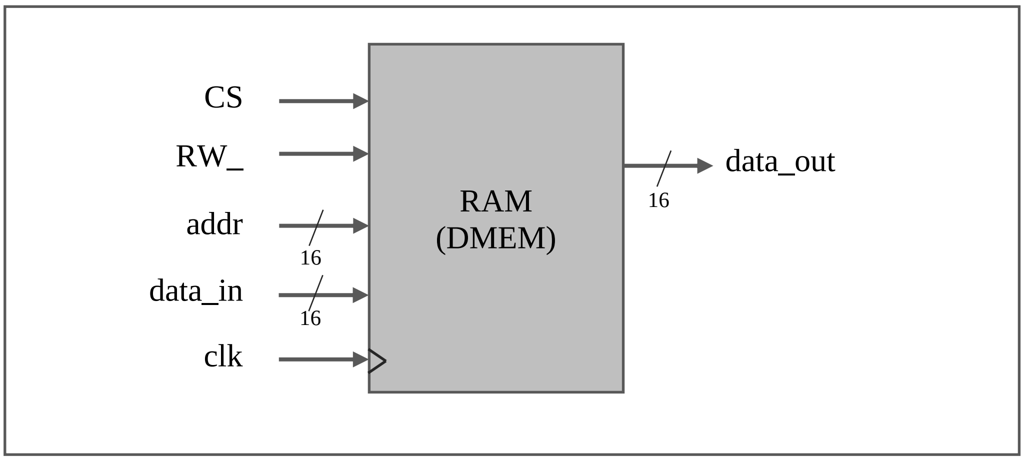


Figure 3.2 : DMEM block diagram

2.4 Register File (RF) Specifications

In addition to ROM and RAM, we also need the register file for our CPU. The RF module specification is as below.

1. The read operations of rs and rt do not depend on clock. As long as address and control signals are valid, the memory contents at the address should be available at the respective output ports

2. The register write operation to rd is synchronous to the clock.

3. Sixteen 16-bit general purpose registers with read and write capability.

4. Inputs:
   Two 4-bit register select for source registers (rs and rt)
   One 4-bit register select for destination register (rd)
   One 16-bit data input (data_in)
   1-bit write enable for rd register (write_en)
   1-bit reset input that will clear the contents of all registers (rst).
5. Outputs:
   Two 16-bit data outputs (rs_data_out and rt_data_out)

Refer to Section 2.5 for details on special registers in this register file.

2.4.1 Special Registers from the RF

1. Register 0 (r0) is a special read-only register with zero value. Writing any value to this register should not change its contents.

2. Register 15 (r15) is a special register, which we would use to display outputs on the LEDs on the DE0-Nano FPGA board. We will use the least significant 8 bits of this register (i.e. r0[7:0] to connect to LED7-LED0, respectively.

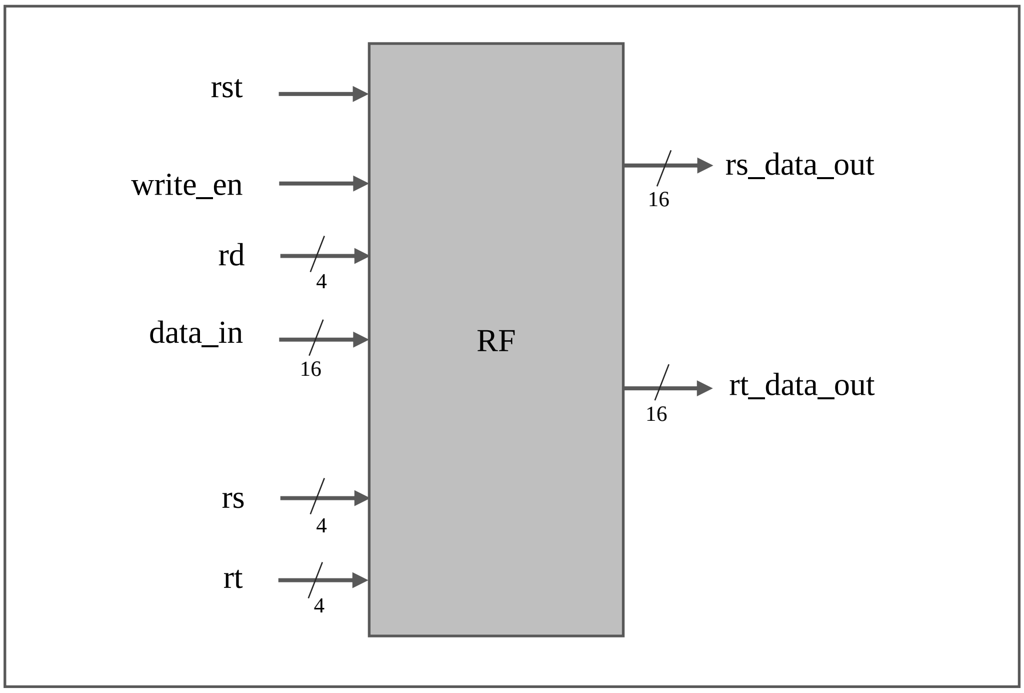
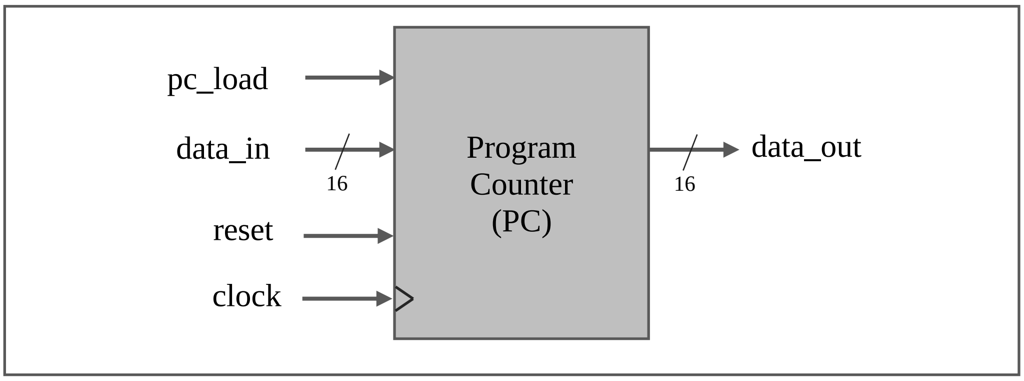


Figure 3.3 : RF block diagram

2.5 Program Counter (PC)

Program counter stores the current address of the instruction being executed. It copies the data_in to data_out at the clock edge only when pc_load signal is high. When reset is high, data_out is initialized to zero.



What you need to do

1. You will work in a team of two.
2. Create each sub-module in Part 2 first and verify their functionalities before proceeding to Part 1.
3. Create another module referred to as test bench to verify the functionality of each sub-module. Thoroughly verify all your sub-modules with your team member.
4. The testbench should test all operators; for each operator, you need to test at all corner cases.
5. For Part 1, there are two things to be considered. First, you need to integrate all the sub-modules as shown in Figure 1.1
6. Then, you need to figure out the controller design on paper first before working on Verilog code for it.
7. Similarly, thoroughly verify the controller using testbench and check all corner cases
8. Once the datapath and controller have been verified, you can integrate them into a top module.
9. To make sure the CPU can be executed, you need to set the program counter to the address of the first instruction in IMEM. For simplicity, you can set this to address 0x0000.
10. You then need to initialize the IMEM ROM with the set of instructions that you want to execute in order to verify that the CPU can execute all the instructions correctly.
11. In order to make it work, you will also need to provide some initial values inside the DMEM for your program to manipulate. This is your verification choice.

What happens after that?

1. You will need to prepare a report on your CPU design and verification process. Demonstrate that the CPU is working as intended.
2. Your instructor will provide a set of data for your IMEM and DMEM in order to run a simple program that blinks the LEDs on the DE0-Nano FPGA.
3. The pin configurations will not be provided. Decide how you want to demonstrate your CPU design.
4. Each team is required to prepare a presentation with PPT slides and a live demo on Quartus II and the DE0-Nano FPGA. The demonstration session will be decided later together.
5. After the demo, you will need to submit the project folder. Zip the whole folder (after you've completed the simulations and generated the .sof file) and submit. The zip must contain all the design and output files.