Custom ARM Cortex M0+ Design specifications

1 Objectives

- To understand the relationship between Instruction Set Architecture (ISA) and Microarchitecture (μA).
- To understand the design process of a complex digital circuit.
- To understand how a basic Microprocessor (μP) works.
- To design and simulate a basic μP based on ARMv6 architecture.
- To understand how the Arithmetic and Logic Unit (ALU) provides a status of the result of each operation.
- To design a simple assembly program.

2 Deadline

23:59 hours on Friday June 9th 2023.

3 Teamwork policy

Teams of 3 or 4 students are allowed.

4 Datapath characteristics

The μP shall have the following characteristics.

- Architecture type: Reduced Instruction Set Computer (RISC).
- Number of addresses in Instruction Memory (IM) is 256.
- Instruction width: 16 bits.
  - Each address in IM stores 16 bits.
- Data width: 16 bits.
  - Each address in Data Memory (DM) stores 16 bits.
- Number of registers in Register File (RF): 16.
- Status Register (SR) shall provide at least zero and negative flags.

5 Background

The block diagram of the μP of Figure 1 has been designed to perform `ADD`, `SUB`, `AND`, `OR`, `LDR`, and `STR` instructions.

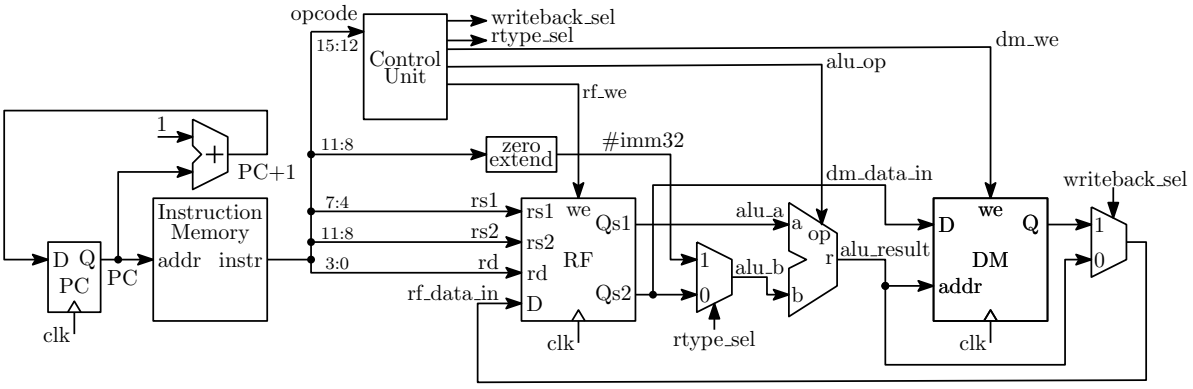

Figure 1: Initial block diagram of a custom ARMv6 μP.

For this assignment, you are required to implement in SystemVerilog the block diagram of Figure 1. In addition to this, you are also required to include the necessary blocks and connections to perform the instructions specified in Section 6.

6 Instruction Set

Table 1 shows the minimum set of instructions that your custom ARMv6 processor should support.

Table 1: Required instructions based on ARMv6 architecture.

Instruction	Syntax	Meaning
	Data processing	
`ADD`	`ADD rd, rs1, rs2`	`Reg[rd] ← Reg[rs1] + Reg[rs2]`
`SUB`	`SUB rd, rs1, rs2`	`Reg[rd] ← Reg[rs1] - Reg[rs2]`
`AND`	`AND rd, rs1, rs2`	`Reg[rd] ← Reg[rs1] & Reg[rs2]`
`OR`	`OR rd, rs1, rs2`	`Reg[rd] ← Reg[rs1] | Reg[rs2]`
`CMP`	`CMP rs1, rs2`	`Reg[rs1] - Reg[rs2]` Updates status register and result is discarded
	Data transfer	
`LDR`	`LDR rd, rs1, #imm`	`Reg[rd] ← Mem[Reg[rs1]+#imm]`
`STR`	`STR rd, rs1, #imm`	`Mem[Reg[rs1]+#imm] ← Reg[rd]`
	Flow control	
`BEQ`	`BEQ label`	`if equal` `then PC ← immediate`
`BMI`	`BMI label`	`if negative` `then PC ← immediate`
`B`	`B label`	Unconditional branch `PC ← immediate`

Table 2 shows the proposed instruction encoding for the processor.

Table 2: Instruction encoding.

	Bit															
	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
`ADD`	0	0	0	0	rs2				rs1				rd			
`SUB`	0	0	0	1	rs2				rs1				rd			
`AND`	0	0	1	0	rs2				rs1				rd			
`OR`	0	0	1	1	rs2				rs1				rd			
`CMP`	0	1	0	0	rs2				rs1				unused			
`LDR`	0	1	1	0	immediate				rs1				rd			
`STR`	0	1	1	1	immediate				rs1				rd			
`BEQ`	1	0	0	0	0	0	0	0	immediate							
`BMI`	1	0	0	0	0	1	0	1	immediate							
`B`	1	0	0	0	1	1	1	1	immediate							

7 Top module and submodules

The following sections detail the minimum required modules for building the custom ARMv6 μP.

7.1 Top

Table 3 describes the required ports of the top level module of the custom ARMv6 μP.

Table 3: Top level port description.

Port name	Direction	Type	Width	Description
clk	input	logic	1	Clock signal
rst_n	input	logic	1	Active low reset signal

7.2 Program Counter

This module indicates which instruction shall be executed within the μP. This is achieved by having a memory element that updates its status at every single clock cycle. The memory element updates its status by either incrementing its previous value or by loading a new value, depending on the instruction to be executed.

The port description of this module is not provided.

7.3 Instruction Memory

Table 4: IM port description.

Port name	Direction	Type	Width	Description
addr	input	logic	$\lceil \log_2(\texttt{IM\_ADDRESSES}) \rceil$	Address port
instr	output	logic	16	Instruction to be executed

7.4 Register File

This block is a memory element that is capable of reading two separate addresses simultaneously. Additionally, this memory element is capable of writing data to the provided write address.

Table 5 provides the description of the inputs and outputs of this block.

Table 5: RF port description.

Port name	Direction	Type	Width	Description
clk	input	logic	1	Clock signal
rst_n	input	logic	1	Active low reset signal
we	input	logic	1	Write enable 0: Read rs1 and rs2. Do not write 1: Read rs1 and rs2. Write data D in address rd
rs1	input	logic	4	Read address 1
rs2	input	logic	4	Read address 2
rd	input	logic	4	Write address
Qs1	output	logic	`DATA_WIDTH`	Outputs data stored in address rs1
Qs2	output	logic	`DATA_WIDTH`	Outputs data stored in address rs2
D	input	logic	`DATA_WIDTH`	Input data for write operations

7.5 Control Unit

This block performs the instruction decoding and enables several control signals for datapath elements. Examples of control signals are:

- MUX selectors
- Write enables
- ALU operation

These control signals are asserted (active) according to the type of instructions to be performed.

The port description of this module is not provided.

7.6 ALU

This block performs arithmetic or logic operations, according to a control signal.

Table 6: ALU port description.

Port name	Direction	Type	Width	Description
alu_op	input	alu_op_t	1	Operation to be performed
a	input	logic	`DATA_WIDTH`	Operand A
b	input	logic	`DATA_WIDTH`	Operand B
z	output	logic	1	Indicates result is exactly 0
n	output	logic	1	Indicates result is negative
r	output	logic	`DATA_WIDTH`	Result from operation

7.7 Data Memory

This is a memory element that stores data generated from instructions. Data is written to this block using `STR` instructions.

Data stored in this memory might later be used by other instuctions. Data is read from this memory using `LDR` instructions.

Table 7: DM port description.

Port name	Direction	Type	Width	Description
clk	input	logic	1	Clock signal
rst_n	input	logic	1	Active low reset signal
we	input	logic	1	Write enable 0: Read from address addr. 1: Write data D to address addr.
D	input	logic	`DATA_WIDTH`	Input data port
addr	input	logic	$\lceil \log_2(\texttt{DM\_ADDRESSES}) \rceil$	Address port
Q	output	logic	`DATA_WIDTH`	Output data port

7.8 Status Register

A Status Register (SR) is a memory element that stores the zero and negative flags generated by the ALU. In each clock cycle, the output from the ALU is provided to the input of the SR, which updates its output only at the rising edge of the clock. The output of the SR is then used by the control unit in conditional branch instructions `BEQ` and `BMI` to determine whether the result of the previous instructions was zero or negative.

Note that Figure 1 does not include any SR and the port description of this module is not provided.

7.9 Package

This is not a block or module. This is a package file containing definitions such as instruction encoding, ALU operations encoding, size of IM and DM, and data width.

Examples of the contents expected in the package file are shown in Listing 1.1 and Listing 1.2.

```
typedef enum logic [1:0] {
  ALU_ADD = 2'b00,
  ALU_SUB = 2'b01,
  ALU_AND = 2'b10,
  ALU_OR  = 2'b11
} alu_op_t;
```

Listing 1.1: alu_op_t definition

```
typedef enum logic [?:0] {
  ADD = ?,
  SUB = ?,
  AND = ?,
  OR  = ?,
  CMP = ?,
  LDR = ?,
  STR = ?,
  BEQ = ?,
```

```
10    BMI = ?,
11    B   = ?
12  } opcode_t;
```

Listing 1.2: opcode_t definition

Note that the `?` symbols in Listing 1.2 must be replaced by their correct values.

8 Deliverables and evaluation

- **[30 points]** A schematic diagram of your μP capable of executing all instructions described in Section 6.
  - All signals must be properly labelled, including names and bit widths.
  - Schematic diagrams generated during Quartus synthesis will not be accepted.
- **[50 points]** All your SystemVerilog design and testbench files.
  - All your files must compile without errors.
  - The testbench file should test the correct operation of each instruction described in Section 6.
  - All instructions should be stored in a text file that is loaded into the IM.
  - The text file contains the machine code for performing each instruction.
- **[20 points]** An assembly code for performing a multiplication of two 8-bit numbers.
  - Assume the operands are stored in `Mem[0]` and `Mem[1]`. The result shall be stored in `Mem[2]`.
  - Assume `Reg[0]=0` and `Reg[1]=1`. These values should help you design your algorithm and its assembly code implementation. It is not necessary to simulate the code, but it is encouraged to validate the correct operation of your algorithm.

9 Submission

Submit your assignment through Canvas **no later than 23:59 hours on Friday June 9th 2023**.

Only one submission per team is required.

IMPORTANT! This is a design exercise. Therefore, there may be **multiple** correct answers.

Please send any questions to isaac.perez.andrade@tec.mx.