Team Red

Allen Liu

Elvis Morales Campoverde

Joye Ma

Xingheng Lin

Rose-Hulman Institute of Technology

Design Document

1 Contents

Table of Contents

Table of Figures

Table of Tables

2 Overview

16-bit Three Stacks Architecture is used in our design. This processor established the stack architecture to simplify the work for the programmer. This processor does not hold a lot of data in the processor itself. Most of the data is stored in the stack in memory. Most algorithms depending on the stack rather than registers. The processor does not contain as many working registers as other processors; it only includes registers reserved for special purposes such as the flag register, interrupt register, and program counter register.

Unlike other processors, this processor will have a minimum size of the instructions loaded into the processor, which downsize the processor's pressure before the program is executed. More space in the main memory will be available for the instructions to use, making this machine more powerful.

With a working stack, a data stack, and a return stack, this processor can separate the argument and return address from the working space and accomplish procedure calls in fewer instructions. The flag register is used for branching and comparing, which is faster than normal instruction.

For double-checking the RTL for errors, the team worked through each RTL on the datapath in multiple separate steps to ensure each step can be done in one cycle.

3 Architecture

3.1 Main Memory Mapping

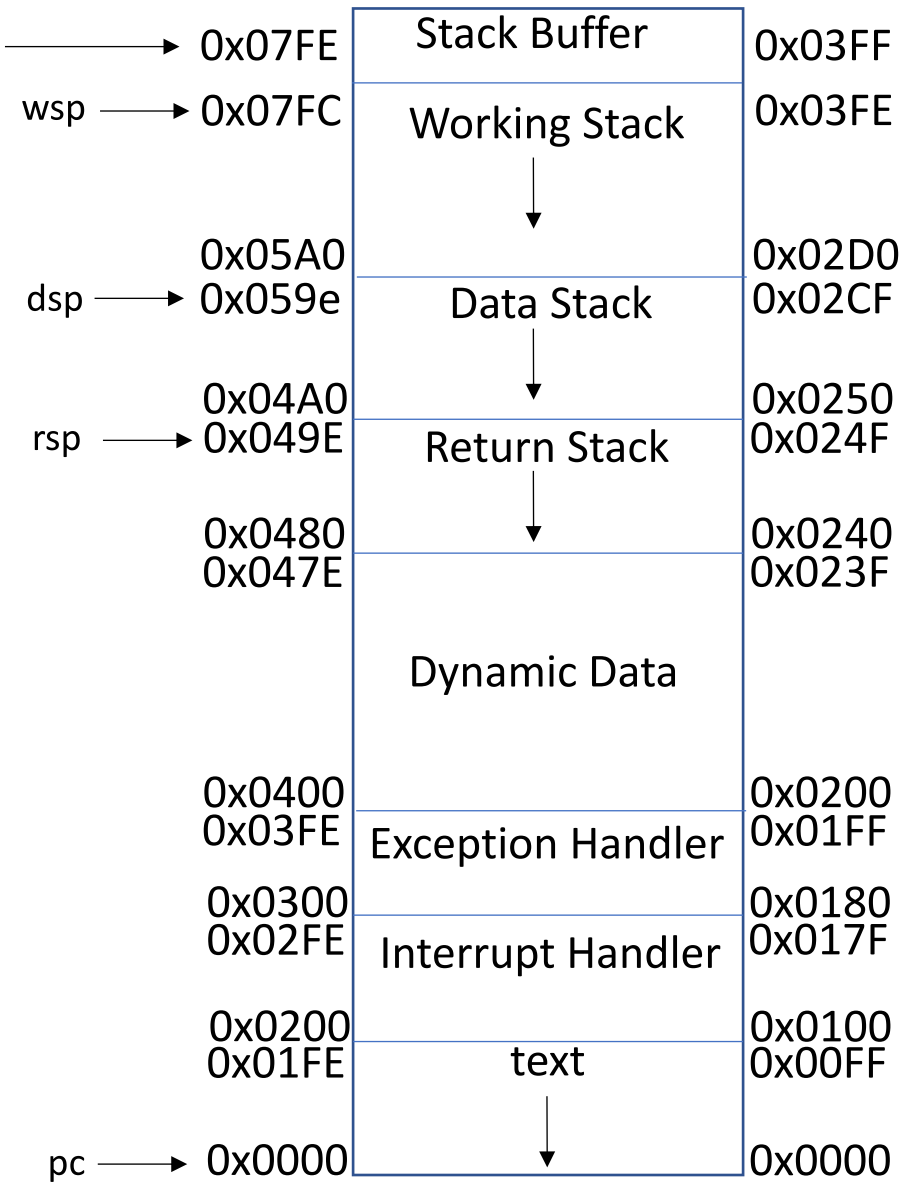

Figure 3-1 Mapping of the Main Memory

3.2 The Three-Stack Architecture

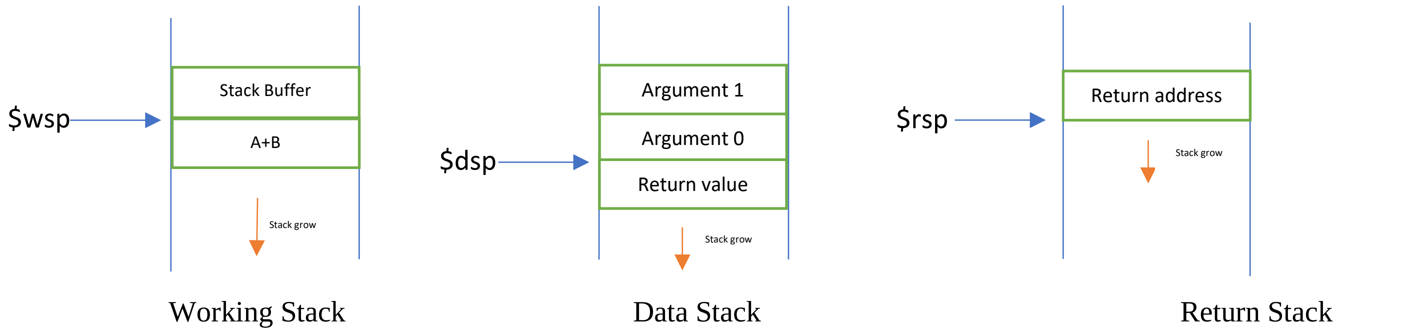

Figure 3-2 Configuration of Three-Stack Architecture

3.3 Datapath

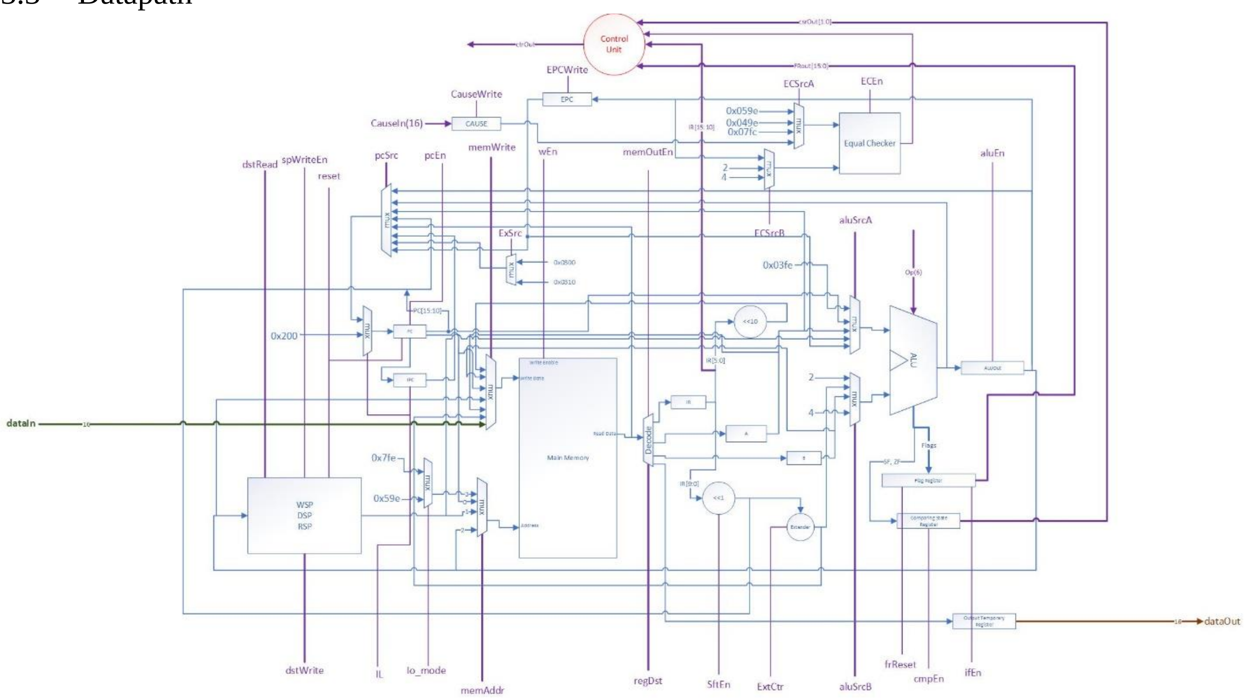

Figure 3-3 Overall DataPath of the processor

3.4 Special Registers

Program Counter Register ($pc)
#Used to keep track of where the next instruction to be executed

Exception Program Counter($epc)
#Used to store the address of the instruction that caused the exception

Interrupt Program Counter($ipc)
#Used to store the address of the pc+2 before going to the interrupt handler

Working stack pointer ($wsp)
#Contains the bottom address of the working stack

Data stack pointer ($dsp)
#Contains the bottom address of the data stack

Return stack pointer ($rsp)
#Contains the bottom address of the return stack

Instruction Register($IR)
#Contains the current instruction

Temporary A Register($A)
#Tempory register that holds the first output from main memory

Temporary B Register($B)
#Tempory register that holds the second output from the main memory

ALUOut Register($ALUOut)
Tempory register that holds the result from ALU

Output Temporary Register($OTR)
#Temporary register that hold the external output

Compare State Reigster($csr)
#store the sign flag and zero flag for later usage

Cause Register($CAUSE)
#store the cause code of exception

- $CAUSE = 2   means arthemetic overflow
- $CAUSE = 4   means stackoverflow

Flag Register ($fr)

Carry	Parity	Sign	Interrupt	Zero	Overflow	Input	Output	X
15	14	13	12	11	10	9	8	7:0

Table 3-1 Flag Register Configuration

- Carry Flag (CF)
    - 0: no carry after addition arithmetic
    - 1: carry after addition arithmetic
- Parity Flag (PF)
    - 0: the result is odd
    - 1: the result is even
- Sign Flag (SF)
    - 0: the result is positive
    - 1: the result is negative
- Interrupt Flag (IF)
    - 0: the interrupt is disabled
    - 1: the interrupt is enabled
- Zero Flag (ZF)
    - 0: the result is not zero
    - 1: the result is zero
- Overflow (OF)
    - 0: no overflow is present in the arithmetic
    - 1: there is an overflow present
- Input (INF)
    - 0: input signal is disabled
    - 1: input signal is enabled
- Output (OUF)
    - 0: output signal is disabled
    - 1: output signal is enabled

3.5 Example Instruction Format

I Type

opcode	Immediate
15 10	9 0

Following the opcode is a 10-bit immediate.

O Type

opcode	X
15 10	9 0

The 10 bits following the opcode are unused.

3.6 Components Specification

3.6.1 Main memory:

- **Signals:**
  - **Address (Addr) (10):**
    - Input signal for reading/writing address to read from/write to a specific location in the main memory.
  - **Reading Data (rD) (16):**
    - Output signal for reading data from the main memory, which is the value stored in the nested block in the main memory.
  - **Write Enable (wEn):**
    - Control signal for controlling whether write is enabled for main memory.
  - **Write Data (wD) (16):**
    - Input signal for the data to be written into the main memory
  - **Clock (CLK):**
    - Clock signal for controlling the dataflow of the entire processor.
- **Description:**
  - When the wEn signal is 1, the main memory will write the value input from wD port into the Addr location of the main memory, and the rD0 and rD1 output value will be same as the value in last clock cycle. When the wEn is 0, the main memory will read the data in Addr and Addr + 2 location and output from rD0 and rD1 port separately.
- **RTL:**
  - **wEn = 1:** M[Addr] = wD
- **Plan to implement**
  - Using ISE Design Suite to generate a Distributed memory unit that has a write width of 16.
- **Plan to test**
  - Use a .coe file to initialize the the first 30 addresses with numbers 0 to 29.
  - Reading will be tested by attempting to read the data we initialized.
  - Writing will be tested by writing an integer into memory, then reading the data to check if the data was written.

3.6.2 Arithmetic Logical Unit (ALU):

- **Signals:**
  - **Operands (A, B) (16):**
    - Input data for ALU computation
  - **Operation (OP) (6):**
    - Control signal for ALU to determine which operation to perform
  - **Output (result) (16):**
    - Result output from ALU
  - **Flags (CF, PF, SF, ZF, OF):**
    - The output signal for setting flags: Carry (CF), Parity (PF), Sign (SF), Zero (ZF) and Overflow (OF).
- **Description:**
  - It will perform the operation controlled by the OP singal and output the result from OUT. It will also output the flag status from the flag ports.

- **RTL:**
  - OUT = A op B
  - CF = Cout[15]
  - PF = !result[0]
  - SF = result[15]
  - ZF = !result[0|:15]
  - OF = Cin[15] ^ Cout[15]
- **Plan to implement**
  1. Build the building blocks of the arithmetic calculation part of the ALU, like the adder, or, and, shifter, etc.
  2. List the operations the ALU supports
  3. Designing the opcode according to the operation list
  4. Wire the building blocks with the multiplexers that selector signal of the multiplexer be the corresponding bit of the opcode
- **Plan to test**
  - Loop through each operation
  - For each operation, run the operation for all possible operands, and test the result

3.6.3 Stack Pointer ($wsp, $dsp, $rsp):

- **Signals**
  - **InputData (spIn) (16):**
    - Input signal for the value to be written into the register
  - **OutputData (spOut) (16):**
    - Output signal for the value read from the register
  - **Clock (CLK):**
    - Clock signal for controlling the process of the entire processor
  - **Write Enable (spWriteEn):**
    - Input signal for controlling whether the value stored in the register could be stored into the register.
  - **Write Destination (dstWrite) (2):**
    - Contro which stack pointer register to write
  - **Read Destination (dstRead) (2):**
    - Contro from which stack pointer resiger to read out data
- **Description:**
  - When spWEn signal is 1, it will overwrite the value in the corresponding stack pointer register indicated by the corresponding wSTD signal by the value of spIn. This component outputs the data read from the register shown by rDST.
- **RTL:**
  - **At the rising edge of the clock**
    - OUT = case(dstRead)
      00: $wsp
      01: $dsp
      10: $rsp
    - **spWriteEn = 1:** case(dstWrite)
      00: $wsp = spIn
      01: $dsp = spIn

10: $rsp = spIn

- **Plan to implement**
    1. Build three registers with the flip-flops
    2. Use a multiplexer to control which data to output
- **Plan to test**
    - Use a sample clock e.g. 50Hz, as the clock signal
    - Read/write one of the resgiter at some random time, and make sure that only at the rising edge of the clock that the value of the register could be changed

3.6.4 Flag Register:

- **Signals:**
    - **Flag Inputs (CFin, PFin, SFin, IFin, ZFin, OFin, EINin, EOUin):**
        - Input signal for setting the flags in the flag register
    - **Outputs (FRout) (16):**
        - Output signal for reading the current flag status from the flag register
    - **Clock (CLK):**
        - Clock signal for reading and writing the data to the register
- **Description:**
    - Value output from OUT port will be the current value stored in the flag register. The value of each bit stored in the flag register will be overwritten as the input value from the corresponding input port.
- **RTL:**
    - **At the rising edge of the clock**
        - **Output:**
            - FRout = FR
        - **Input:**
            - **FR[12] (IF) = 0:**
                - FR[15] = CFin
                - FR[14] = PFin
                - FR[13] = SFin
                - FR[12] = IFin
                - FR[11] = ZFin
                - FR[10] = OFin
                - FR[9] = EINin
                - FR[8] = EOUin
- **Plan to implement**
    - Connect the bits of Flag Register to the corresponding output from ALU
- **Plan to test**
    - Perform subtraction, addition and shift operation and check the corresponding flags such as CarryOut and Zero flag

3.6.5 Extender:

- **Signals:**
    - **Control Input (ExtCtr):**

        - For controlling the function in the extender (0 for zero extender, 1 for sign extender).
    - **Input (IN) (10):**
        - Input signal for the immediate value input.
    - **Output (OUT) (16):**
        - Output signal for the extended value.
- **Description:**
    - This block will sign extend (ExtCtr = 1) or zero extend (ExtCtr = 0) the input value and output from the OUT port
- **RTF:**
    - **ExtCtr = 0:** OUT = {6{1'b0}, IN}
    - **ExtCtr = 1:** OUT = {6{IN[9]}, IN}
- **Plan to implement**
    - Build the extender by connecting the ExtCtr bit with 6 AND gate. Combine the output of these 6 AND gates with the 10 inputs.
- **Plan to test**
    - Test the sign extend by inputing the number and ExtCtr = 1 and check the result. Test the zero extend by inputing the number and ExtCtr = 0

3.6.6 Bit Shifter:

- **Signals:**
    - **Input (a) (16):**
        - Input port for the value to be shifted by the shifter
    - **Output (O) (16):**
        - Output port for output shifted value
    - **Bit Shifter enable (sftEn):**
        - Control whether the bit shifter will perform shift operation at this moment
- **Description:**
    - The OUT will always be the value of IN left shifted by 1
- **RTL:**
    - **sftEn = 1:** O = a << 1
    - **sftEn = 0:** O = a
- **Plan to implement**
    - Connect the [14:0] bits of the input value directly to the [15:1] bits of the output, then connect the last bit of output to the ground.
- **Plan to test**
    - Test by putting a 1 on every bit and see if it is shifted to the one higher bit without changing other 0's.

3.6.7 Functional Registers ($ALUOut, $A, $B, $IR, $PC,$EPC,$IPC, $OTR,$CAUSE):

- **Signals:**
    - **Input (IN) (16):**
        - Input signal for the value to be written into the register
    - **Output (OUT) (16):**
        - Output signal for the value read from the register

- **Clock (CLK):**
  - Clock signal for controlling the process of the entire processor
- **Write Enable (wEn):**
  - Input signal for controlling whether the value stored in the register could be stored into the register.

- **Description:**
  - Output will be set to the input when CLK is at rising edge and write enable is 1
- **RTL:**
  - **At the risng edge of the clock**
    - **wEn = 1:** OUT = IN
    - **wEn = 0:** OUT = $xxx (name of the register)
- **Plan to implement**
  - These registers will be constructed with flip-flops
  - Each register will have a wEn controlling bit
- **Plan to test**
  - Test each register by writing 0 and 1 to it, in both conditions in which the wEn bit is 0 and 1, then check the output of the registers.

3.7 Inplementation plan

- Stage 1: implement each component based on the RTL and perform the unit test on the component separately [Done]
- Stage 2: Connect the components to subdatapath and test it[Done]
  - ALU with Flag register
  - Main memory with mux
  - Fetch parts
  - Stack pointers
- Stage 4: Connect these four subdatapaths into one piece [Done]
- Stage 5: Write system unit tests to test the complete datapath on each instruction. The system unit test covers the usage of every path of the complete datapath. [Done]
- Stage 6: Write a test for control units before connecting to datapath control signal [Done]
- Stage 7: Connect the control signal to control units [Done]
- Stage 8: Unit tests for the complete processor with both datapath and control [Done]

3.8 Control testing plan

We plan to write the control in Verilog firstly. Then we input the opcode, which should come from IR[15:10], manually to control the unit and write the test to check if the controller outputs the proper signal in each stage.

3.9 Implementation

- Memory implementation: See Appendix 12.1
- ALU, Flag Register implementation: See Appendix 12.2
- Fetch part, extender, temporary register implementation: See Appendix 12.3
- Full Datapath: See Appendix 12.4
- Exception Part: See Appendix 12.5
- Interrupt Part: See Appendix 12.6

3.10 Instruction Description

- Add – add (O - type)
  - Add the top two elements of the working stack and store the result back
- Add Immediate – addi (I - type)
  - Add the top of the working stack with sign-extended immediate and store the result back
- And – and (O - type)
  - And the top two elements of the working stack and store the result back
- And Immediate – addi (I - type)
  - And the top of working stack with zero-extended immediate and store the result back
- Branch if less than – blt (O - type)
  - Branch if the result from compare is "less than," which is stored in the "Sign" flag from flag register($fr)
- Branch if greater than – bgt (O - type)
  - Branch if the result from compare is "greater than," which can be checked in the "Sign" and "Zero" flags in the flag register ($fr)
- Branch if equal – beq (O - type)
  - Branch if the result from compare is "equal," which is obtained from "Zero" flags in flag register ($fr)
- Call procedure– call (I - type)
  - save the $pc + 2 to return stack and call the procedure
- Clear Working Stack– clr (I - type)
  - Move the wsp back based on the offset to achieve the functionality of clearing the top few elements from the working stack
- Compare – cmp (O - type)
  - Compare the top two elements of the working stack and set the flag of the flag register
- Jump – j (I - type)
  - Jump to the address from the jump instruction
- Jump stack– js (O- type)
  - Jump to the address that is stored at the top of the working stack
- Load from data stack– ld (I- type)
  - Load the element from the data stack, which can either be an argument or return value, to working stack based on the offset
- Load Upper Immediate– lui (I- type)
  - Load upper 8 bit immediate to a working stack
- Move Data Stack Pointer – mdsp (I- type)
  - Move the data stack pointer by offset, which is used for allocating argument space for procedure call or restoring the data stack after the procedure call
- Or Immediate - ori (I - type)
  - Or operation between the top of the working stack and Immediate
  - Store the result back by overwriting the top of the working stack
- Or - or (O - type)
  - Or operation between the top two elements of the working stack
  - Store the result back by overwriting the top two elements with the result
- Pop to Main Memory- pop (I - type)
  - Pop the top element of the working stack to the destination address
- Push immediate – pushi (I - type)

    - Push Immediate value to a working stack
- Push from Main Memory – push (I - type)
    - Push the element from the destination address to the top of the working stack
- Return from Procedure Call – ret (O - type)
    - Return from procedure call by restoring the program counter from the return stack
- Shift left – sfl (I - type)
    - Left shift the top of the working stack based on the immediate offset
- Shift right – sfr (I - type)
    - Right shift the top of the working stack based on the immediate offset
- Store to data stack – st (I - type)
    - Store the top of the working stack to data stack with offset, which can be used to prepare the arguments for a procedure call
- Subtraction – sub (O - type)
    - Subtract the top element of the working stack from the second to top element of the working stack
    - Store the result back by overwriting the first two-element
- Swap top two elements – swap (O - type)
    - Swap the top two elements of the working stack
- Exit- exit (O - type)
    - Swap the top two elements of the working stack

3.11 Instruction Set

3.11.1 Fundamental Instruction Set

Name	Mnemonic	Format	Operation	Opcode
Add	`add`	O	M[$wsp] = M[$wsp+2]+ M[$wsp] $wsp = $wsp + 2	100000
Add Immediate	`addi`	I	M[$wsp] = M[$wsp] + SignExtImm	100001
And	`and`	O	M[$wsp] = M[$wsp] & M[$wsp + 2] $wsp = $wsp + 2	100010
And Immediate	`andi`	I	M[$wsp] = M[$wsp] & ZeroExtImm	100100
Branch if less than	`blt`	I	(SF == 1) ? $pc = $pc + 2 + BranchAddr : $pc = $pc +2	101000
Branch if greater than	`bgt`	I	(SF == 0 && ZF == 0) ? $pc = $pc + 2 + BranchAddr : $pc = $pc +2	110000
Branch if equal	`beq`	I	(ZF == 1) ? $pc = $pc + 2 + BranchAddr : $pc = $pc +2	110001
Call Procedure	`call`	I	$rsp = $rsp – 2 , M[$rsp] = $pc + 2, $pc = CalleeAddr	110010
Compare	`cmp`	O	if M[$wsp + 2] > M[$wsp] : SF = 0, ZF = 0 else if M[$wsp + 2] == M[$wsp]: ZF = 1 else if M[$wsp + 2] < M[$wsp]: SF = 1, ZF = 0 $wsp = $wsp + 4	111010
Clear Working Stack	`clr`	I	$wsp = $wsp + (SignExtImm<<1)	110111
Exit	`exit`	O		000000
Jump	`j`	I	$pc = JumpAddr	101010
Jump Stack	`js`	O	$pc = M[$wsp], $wsp = $wsp + 2	101110
Load from data stack	`ld`	I	$wsp = $wsp – 2, M[$wsp] = M[$dsp + SignExtImm<<1]	101111
Load Upper Immediate	`lui`	I	$wsp = $wsp – 2, M[$wsp] = Upper8bit(Imm) 8’b0	100101
Move (Data) Stack Pointer	`mdsp`	I	$dsp = $dsp + Imm<<1	100110
Or Immediate	`ori`	I	M[$wsp] = M[$wsp] | ZeroExtImm	000001
Or	`Or`	O	M[$wsp] = M[$wsp] | M[$wsp + 2] $wsp = $wsp + 2	000010
Pop to Main Memory	`pop`	I	M[MemAddress] = M[$wsp], $wsp = $wsp + 2	000100
Push Immediate	`pushi`	I	$wsp = $wsp – 2, M[$wsp] = SignExtImm	000110
Push from Main Memory	`push`	I	M[$wsp], $wsp = $wsp – 2, M[$wsp] = M[MemAddress]	001000
Return from Procedure Call	`ret`	O	$pc = M[$rsp], $rsp = $rsp + 2	001010
Shift Left	`sfl`	I	M[$wsp] = M[$wsp] << Imm	001110
Shift Right	`sfr`	I	M[$wsp] = M[$wsp] >> Imm	010000

Store to data stack	`st`	I	M[$dsp + SignExtImm<<1] = M[$wsp] $wsp = $wsp + 2	010010
Subtract	`sub`	O	M[$wsp] = M[$wsp+2] - M[$wsp] $wsp = $wsp + 2	010011
Swap top two element	`swap`	O	M[$wsp-2] = M[$wsp], M[$wsp] = M[$wsp + 2], M[$wsp + 2] = M[$wsp - 2]	011000

Table 3-2 Table of the Instruction Set

(1) May cause overflow exception
(2) SignExtImm = { 6{immediate[9]}, immediate }
(3) ZeroExtImm = { 6{1'b0}, immediate}
(4) BranchAddr = { 5{immediate[9]}, immediate, 1'b0 }
(5) CalleeAddr = { PC+2[15:11], immediate, 1'b0}
(6) JumpAddr = { PC+2[15:11], immediate, 1'b0}
(7) MemAddress = { PC+2[15:11], immediate, 1'b0}

3.11.2 Instruction Set for Interrupt Handler

Name	Mnemonic	Format	Operation	Opcode
Load input to Stack Buffer	`ldinbf`	O	SB = InputData	111100
Return from interrupt handler to main program	`iret`	O	PC = IPC	111101

Table 3-3 Table of Instruction Set for Interrupt Set

3.11.3 Instruction Set for Exception Handler

Name	Mnemonic	Format	Operation	Opcode
Intentify exception cause	`iexc`	O	If (cause == 2) PC = AOFExceptionAddr If (cause == 4) PC = SOFExceptionAddr	111000
Return from exception handler to main program	`eret`	O	PC = EPC	111001
Ignore current instruction	`ignr`	O	PC = EPC + 2	111010

Table 3-4 Table of Instruction Set for Exception Handler

3.11.4 Step Chart_1

Step	Add/Sub/Or/And	Addi/Andi/Ori	cmp	J	Call	bgt/bst/beq	js
Inst Fetch	IR = M[PC] PC = PC + 2						
Inst Decode Load A	A = M[$wsp] ALUOut = $wsp + 2						
wsp dec_1	B = M[ALUOut] $wsp = ALUOut						
Load B ALU Execuation	ALUOut = A op B	ALUOut = A op SE(IR[9:0])	B = M[ALUOut] ALUOut = $wsp + 4	PC = PC[15:11] || [IR[9:0] << 1]	ALUOut= rsp – 2	ALUOut=PC + SE[IR[9:0] << 1]	PC =A $wsp = ALUOut
Mem Write	M[$wsp] = ALUOut	M[$wsp] = ALUOut	$wsp = ALUOut A – B (Set Flag)		M[ALUOut] = PC PC = PC[15:11] || [IR[9:0] << 1] $rsp = ALUOut	If (Flag): PC = ALUOut	

Flag:

A > B: Reg[$fr][13] = 0, Reg[$fr][11] = 0

A == B: Reg[$fr][11] = 1

A < B: Reg[$fr][13] = 1, Reg[$fr][11] = 0

3.11.5 Step Chart_2

Step	Swap	st	**pop**	push	ld	lui	pushi
Inst Fetch	IR = M[PC] PC = PC + 2						
Inst Decode Load A	A = M[$wsp] ALUOut = $wsp + 2						
wsp dec_1				ALUout=$wsp – 2			
Load B ALU Execution	B =M[ALUOut] M[ALUOut] = A	$wsp = ALUOut ALUOut = $dsp + SE(IR[9:0]<<1)	$wsp = ALUOut ALUOut = #constant+ZE(IR[9:0] << 1)	$wsp = ALUOut ALUOut = #constant+ZE(IR[9:0] << 1)	$wsp = ALUOut ALUOut = $dsp + SE(IR[9:0] << 1)	$wsp = ALUOut	$wsp = ALUOut
Mem Write	M[wsp] = B	M[ALUOut] = A	M[ALUOut] = A	A = M[ALUOut]	A= M[ALUOut]	M[$wsp]= (IR[5:0]<<10)	M[$wsp] = SE(IR[9:0])
				M[$wsp] = A	M[$wsp] = A		

3.11.6 Step Chart_3

Step	sfl/sfr	iexc	mdsp	ret	clr	exit	ldinbf	iret
Inst Fetch	IR = M[PC] PC = PC + 2							
Inst Decode Load A	A = M[$wsp] ALUOut = $wsp + 2							
wsp dec_1								
Load B ALU Execution	ALUOut= A << or >> SE(IR[9:0])	if (CAUSE == 2): PC = AOFExceptionAddr else if (CAUSE == 4): PC = SOFExceptionAddr	ALUOut= $dsp + (IR[9:0]<<1)	PC = M[$rsp] ALUOut= $rsp + 2	ALUOut= $wsp + (IR[9:0]<<1)	DataOutput = M[0x59e]	M[SB] = DataInput	PC = IPC
Mem Write	M[$wsp] = ALUOut		$dsp = ALUOut	$rsp = ALUOut	$wsp=ALUOut			

Table 3-5 Multicycle RTL of the processor

4 Multicycle Control

Controlled signals:

- dstRead [1:0]: controlling the stack pointer to be read
- spWriteEn: enabling the write to any stack pointer
- dstWrite [1:0]: controlling the stack pointer to be written
- pcSrc [2:0]: mux for sorce of pc
- pcEn: controlling the write to the program counter
- memWrite [2:0]: controlling data from which source will be written into the memory
- memAddr [1:0]: mux for the address of the data in memory
- wEn: enableing memory write
- memOutEn: enabling memory output
- regDst [1:0]: controlling to which register the memory output is written to
- SftEn: controlling if bit shifting is allowed
- ExtCtr: controlling the mode of extender
- aluSrcA [1:0]: mux for the source of alu operand a
- aluSrcB [1:0]: mux for the source of alu operand b
- aluOP [5:0]: opcode of the alu used to select the operation performed in alu
- aluEn: controlling if written to ALUOut register is allowed
- cmpEn: controlling the write to a 2-bits register that store the sign and zero flags

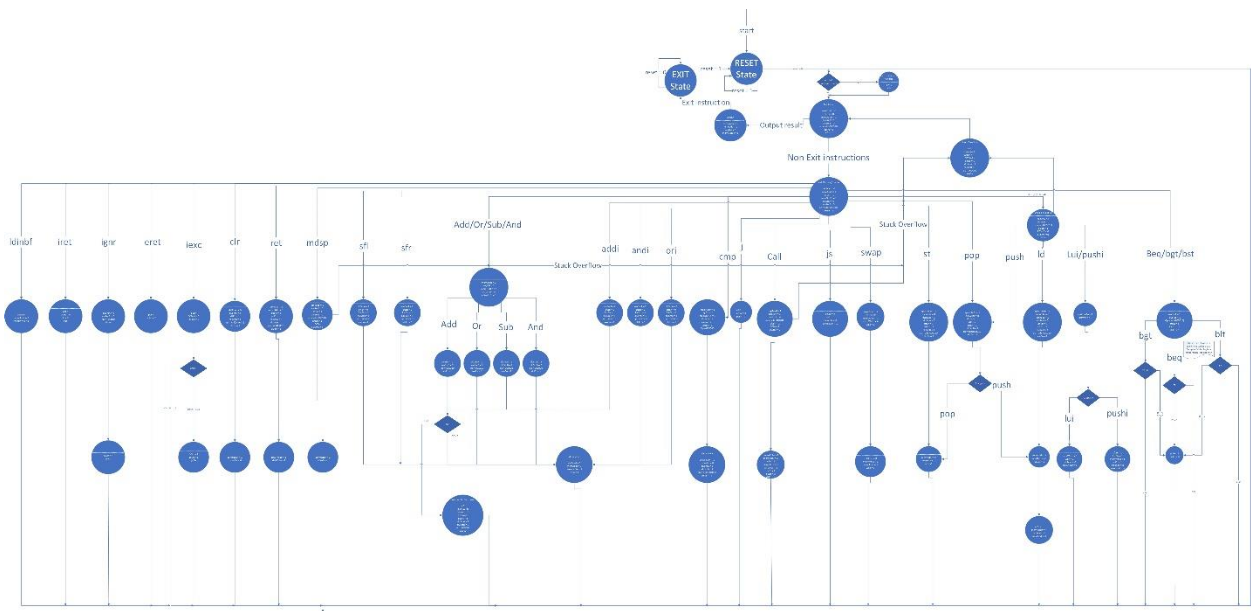

Figure 4-1 Multicucle Control FSM

5 Procedure Call Convention

Before the procedure is called, the caller should do the following:

1. Set up arguments by pushing arguments into the data stack frame
2. Push the return address into the return stack frame
3. Call procedure by call, which put the return address into the return stack

Before the callee's body is executed, the callee should do the following:

1. Reserve space in the data stack for any local variables in the function
2. Execute the body

When the body of the callee finished execution, the callee should do the following:

1. Restore the data stack pointer as before the procedure has been called
2. Put the return value into the block just below the data stack pointer
3. Return to the caller using the return address stored into the return stack, and restore the return stack pointer

After returning to the caller, the caller should do the following:

1. Get the returned value from the block below the data stack pointer

6 Input/Output

6.1 Input

When the user would like to let the processor to take the input, user would first prepare the input, after the input value is ready, user will trigger the external interrupt and the interrupt handler will trigger the input and it will read the value from the input port and store it into the stack buffer.

6.2 Output

After the program finished executing, it will read the resukt, which is the value stored in the top of the data stack, and write that value into the output resgiter, when the user can read the value from the output port.

7 Exception

Two types of exceptions are designed and implemented. One is the arithmetic overflow and the other is the stack overflow. The exception types and their corresponding cause code are shown in the table below.

Exception Type	Cause Code
Arithmetic Overflow	2
Stack Overflow	4

The control unit will constantly check the value of the overflow flag bit (which keeps tracks of the arithmetic overflow happened in the ALU) of the flag register. It will guide PC to the exception handler insofar as it receives a HIGH signal of the overflow flag.

When the instructions which involve stack pointer subtraction (i.e., request to use more stack space) are executed, which in our case are `call`, `ld`, `push`, `pushi`, `lui`, and `mdsp`, the equal checker checks the validity of the latest stack pointer value calculated by ALU, and if the new stack pointer value reaches the limit of that stack, the program will be directed to the exception handler.

Three new instructions were added to the design for exception handling.

Mnemonic	Description
`iexc`	Load and identify the cause code stored in the cause register, then direct the program to the specific handling codes.
`ignr`	Ignore the exception and return to and run the next line of the instruction that caused the exception.
`eret`	Return to the instruction that caused the exception and retry.

The first instruction of the exception handler is `iexc`. It retrieves cause codes and sets PC to the addresses that store certain handling codes. For Arithmetic Overflow, the processer will ignore this exception and return to running the next instruction of the original code. The program will be terminated if a Stack Overflow exception occurs.

8 Interrupt

for the external interrupt, to trigger it,user will enable the input signal for the interrupt input, and the external interrupt in the processor will be triggered, and the processor will automatically jump to the interrupt handler portion of the instruction, and execute the instructions for interrupt handler. After the processor finished executing the instructions for the interrupt handler, it will automatically jump back to the next instruction to be executed in the main program.

9 Example assembly language program

```
        MAIN:
0x0000          j       MAIN        #Start from the main program
0x0002          push    0x200       #Push external input stored in Stack
0x0004          st      0           #set up the argument for relprime
0x0006          call    RELPRIME    #call RELPRIME function
0x0008          ld      -1          #load result from RELPRIME
0x000a          exit                #exit after the Main program is finished
        RELPRIME:
0x000c          mdsp    -1          #move the data stack pointer down one block  for m
0x000e          pushi   2           #push immediate 2 to working stack
0x0010          st      0           #store 2 from the top of working stack to data stack
        LOOP_1:
0x0012          mdsp    -2          #move the data stack pointer down two block
0x0014          ld      2           #load m from data stack to working stack
0x0016          st      1           #store m back to data stack as argument 1
0x0018          ld      3           #load n from data stack to working stack
0x001a          st      0           # store n back to data stack as argument 0
0x001c          call    GCD         #procedure call GCD
0x001e          ld      -1          #load the return value of GCD to working stack
0x0020          pushi   1           #push 1 to working stack
0x0022          cmp                 #compare if the return value is equal to 1 and set the flag
0x0024          beq     DONE        #branch based on the flag
0x0026          ld      0           #load m from data stack to working stack
0x0028          addi    1           #add 1 to m at working stack
```

```
0x002a        st     0        #store m back to data stack
0x002c        j      LOOP_1   #Jump to loop1
    DONE:
0x002e        ld     0        #load the m to working stack
0x0030        mdsp   2        #free the data stack
0x0032        st     -1       #store the return value m
0x0034        ret             #return
    GCD:
0x0036        ld     0        #load the a to working stack
0x0038        pushi  0        # push 0 to stack
0x003a        cmp             # a ? 0
0x003c        beq    RET_B
    LOOP_2:
0x003e        ld     1        # load b to working stack
0x0040        pushi  0        # push 0 to working stack
0x0042        cmp             # b ? 0
0x0044        beq    RET_A    #branch to return_A
0x0046        ld     1        # load b to working stack
0x0048        ld     0        # load a to working stack
0x004a        cmp             # a ? b
0x004c        bgt    ELSE     #branch to ELSE
0x004e        ld     0        # load b to working stack
0x0050        ld     1        # load a to working stack
0x0052        sub             # top(working stack) = b – a
0x0054        st     1        #b = top(working stack)
0x0056        j      END
    ELSE:
0x0058        ld     1        # load a to working stack
0x005a        ld     0        # load b to working stack
0x005c        sub             # top(working stack) = a – b
0x005e        st     0        #b = top(working stack)
```

```
        END:
0x0060          j     LOOP_2
        RET_A:
0x0062          ld    0         # load a to working stack
0x0064          mdsp  2         # move the data stack pointer back
0x0066          st    -1        # store return value a from working stack to data stack
0x0068          ret             # return to caller
        RET_B:
0x006a          ld    1         # load b to working stack
0x006c          mdsp  2         # move the data stack pointer back
0x006e          st    -1        # store return value b from working stack to data stack
0x0070          ret             # return to caller
```

10 Assembly Language Fragments

OPERATION	C CODE	ASSEMBLY CODE	MACHINE CODE TRANSLATION

RECURSION	if n = 1
	return 1;
	else
	return n + sum(n - 1);
		SUM:
		ld 0
		pushi 1
		sub
		beq END
		mdsp -1
		st 0
		call SUM
		ld -1
		ld 0
		add
		st 0
		j EXIT
		END:
		clr 1
		EXIT:
		mdsp 1
		ret
			
			0b101111 0000000000
			0b000100 0000000001
			0b010100 0000000000
			0b110001 ENDAddr
			0b100110 1111111111
			0b010010 0000000000
			0b110010 SUMAddr
			0b101111 1111111111
			0b101111 0000000000
			0b100000 0000000000
			0b010010 0000000000
			0b101010 EXITAddr
			
			0b110111 0000000001
			
			0b100110 0000000001
			0b001010 0000000000impl

ITERATION	for (i = 0; i <= 10; i++) {
	// something in
	}
	// something out
		LOOP:
		push i
		pushi 10
		cmp
		bgt OUT
		// something in
		j LOOP
		OUT:
		// something out
			
			0b001000 i_addr
			0b000110 0000001010
			0b111010 0000000000
			0b110000 OUTAddr
			
			0b101010 LOOPAddr

CONDITIONAL BRANCH	If (a <= b) {
	// something
	}
		push a
		push b
		cmp
		blt DONE
		// something
		DONE:
			0b001000 a_addr
			0b001000 b_addr
			0b111010 0000000000
			0b101000 DONEAddr

READ FROM INPUT PORT	read() // read from input port	ein	0b111101 0000000000
DISPLAY	display() // display the value stored in the top of the stack	top eou	0b011100 0000000000 0b111111 0000000000

Table 10-1 Sample assembly language of the processor

// I/O instructions only controls the data flow between the memory and the I/O port, the actual process of the I/O will be implementes by the external interrups.

11 Machine Language Translation

11.1 Main Program

MAIN:				
0x0000	j	MAIN	0b101010 0000000000	0xa400
0x0002	push	0x200	0b001000 1000000000	0x2200
0x0004	st	0	0b010010 0000000000	0x4800
0x0006	call	RELPRIME	0b110010 0000000110	0xc806
0x0008	ld	-1	0b101111 1111111111	0xbfff
0x000a	exit		0b000000 00000000000	0x0000
RELPRIME:				
0x000c	mdsp	-1	0b100110 1111111111	0x9bff
0x000e	pushi	2	0b000110 0000000010	0x1802
0x0010	st	0	0b010010 0000000000	0x4800
LOOP_1:				
0x0012	mdsp	-2	0b100110 1111111110	0x9bfe
0x0014	ld	2	0b101111 0000000010	0xbc02
0x0016	st	1	0b010010 0000000001	0x4801
0x0018	ld	3	0b101111 0000000011	0xbc03
0x001a	st	0	0b010010 0000000000	0x4800
0x001c	call	GCD	0b110010 0000011011	0xc81b
0x001e	ld	-1	0b101111 1111111111	0xbfff
0x0020	pushi	1	0b000110 0000000001	0x1801
0x0022	cmp		0b111010 0000000000	0xe800
0x0024	beq	DONE	0b110001 0000000100	0xc404
0x0026	ld	0	0b101111 0000000000	0xbc00
0x0028	addi	1	0b100001 0000000001	0x8401
0x002a	st	0	0b010010 0000000000	0x4800
0x002c	j	LOOP_1	0b101010 0000001001	0xa809
DONE:				
0x002e	ld	0	0b101111 0000000000	0xbc00

0x0030	mdsp	2	0b100110 0000000010	0x9802
0x0032	st	-1	0b010010 1111111111	0x4bff
0x0034	ret		0b001010 0000000000	0x2800
GCD:				
0x0036	ld	0	0b101111 0000000000	0xbc00
0x0038	pushi	0	0b000110 0000000000	0x1800
0x003a	cmp		0b111010 0000000000	0xe800
0x003c	beq	RET_B	0b110001 0000010110	0xc416
LOOP_2:				
0x003e	ld	1	0b101111 0000000001	0xbc01
0x0040	pushi	0	0b000110 0000000000	0x1800
0x0042	cmp		0b111010 0000000000	0xe800
0x0044	beq	RET_A	0b110001 0000010110	0xc40e
0x0046	ld	1	0b101111 0000000000	0xbc01
0x0048	ld	0	0b101111 0000000001	0xbc00
0x004a	cmp		0b111010 0000000000	0xe800
0x004c	bgt	ELSE	0b110000 0000000101	0xc005
0x004e	ld	0	0b101111 0000000000	0xbc00
0x0050	ld	1	0b101111 0000000001	0xbc01
0x0052	sub		0b010011 0000000000	0x4c00
0x0054	st	1	0b010010 0000000001	0x4801
0x0056	j	END	0b101010 0000110000	0xa830
ELSE:				
0x0058	ld	1	0b101111 0000000001	0xbc01
0x005a	ld	0	0b101111 0000000000	0xbc00
0x005c	sub		0b010011 0000000000	0x4c00
0x005e	st	0	0b010010 0000000000	0x4800
END:				
0x0060	j	LOOP_2	0b101010 0000011111	0xa81f
RET_A:				
0x0062	ld	0	0b101111 0000000000	0xbc00

0x0064	mdsp	2	0b100110 0000000010	0x9802
0x0066	st	-1	0b010010 1111111111	0x4bff
0x0068	ret		0b001010 0000000000	0x2800
RET_B:				
0x006a	ld	1	0b101111 0000000001	0xbc01
0x006c	mdsp	2	0b100110 0000000010	0x9802
0x006e	st	-1	0b010010 1111111111	0x4bff
0x0070	ret		0b001010 0000000000	0x2800

11.2 Interrupt Handler

0x0200	ldinbf		0b111100 0000000000	0xf000
0x0202	iret		0b111101 0000000000	0xf400

11.3 Exception Handler

0x0300	iexc		0b111000 0000000000	0xe000
0x0302	ignr		0b111010 0000000000	0xe800
0x0310	exit		0x000000 0000000000	0x0000

12 Appendix

12.1 Subsystem for Main Memory

12.1.1 Main memory subsystem

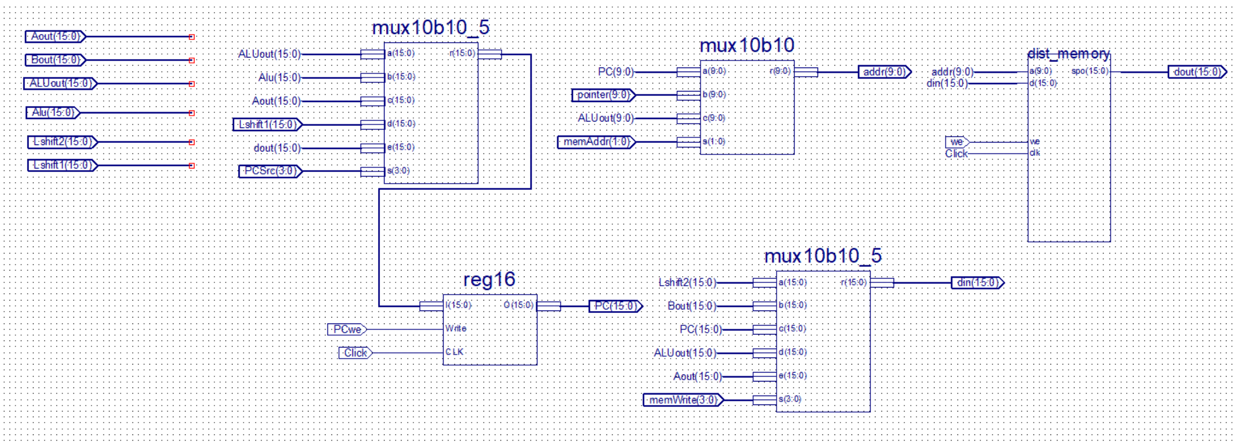

Figure 12-1 Main Memory and PC Subsystem

12.2 Subsystem for ALU, ALUout and Flag Register

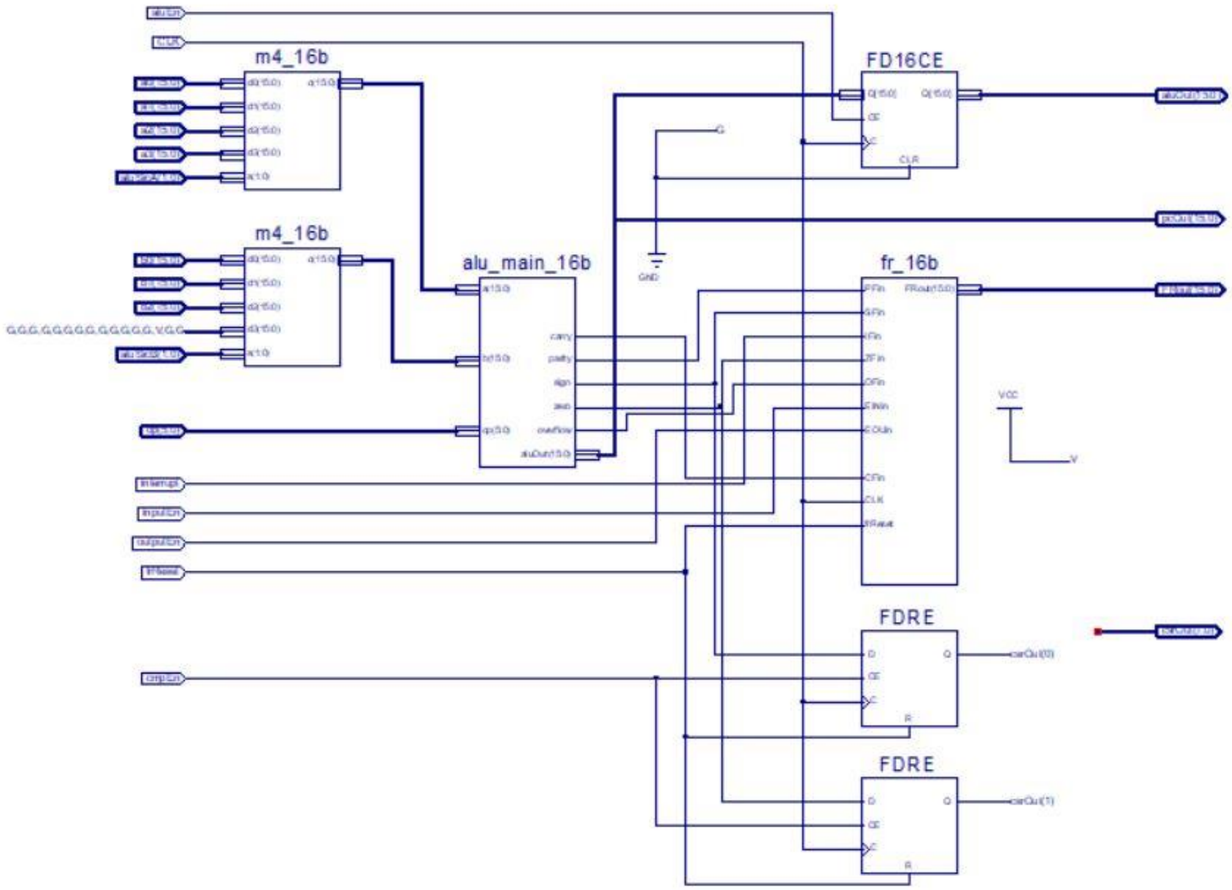

Figure 12-2 ALU, ALUOut and Flag Register Subsystem

12.2.1 ALU

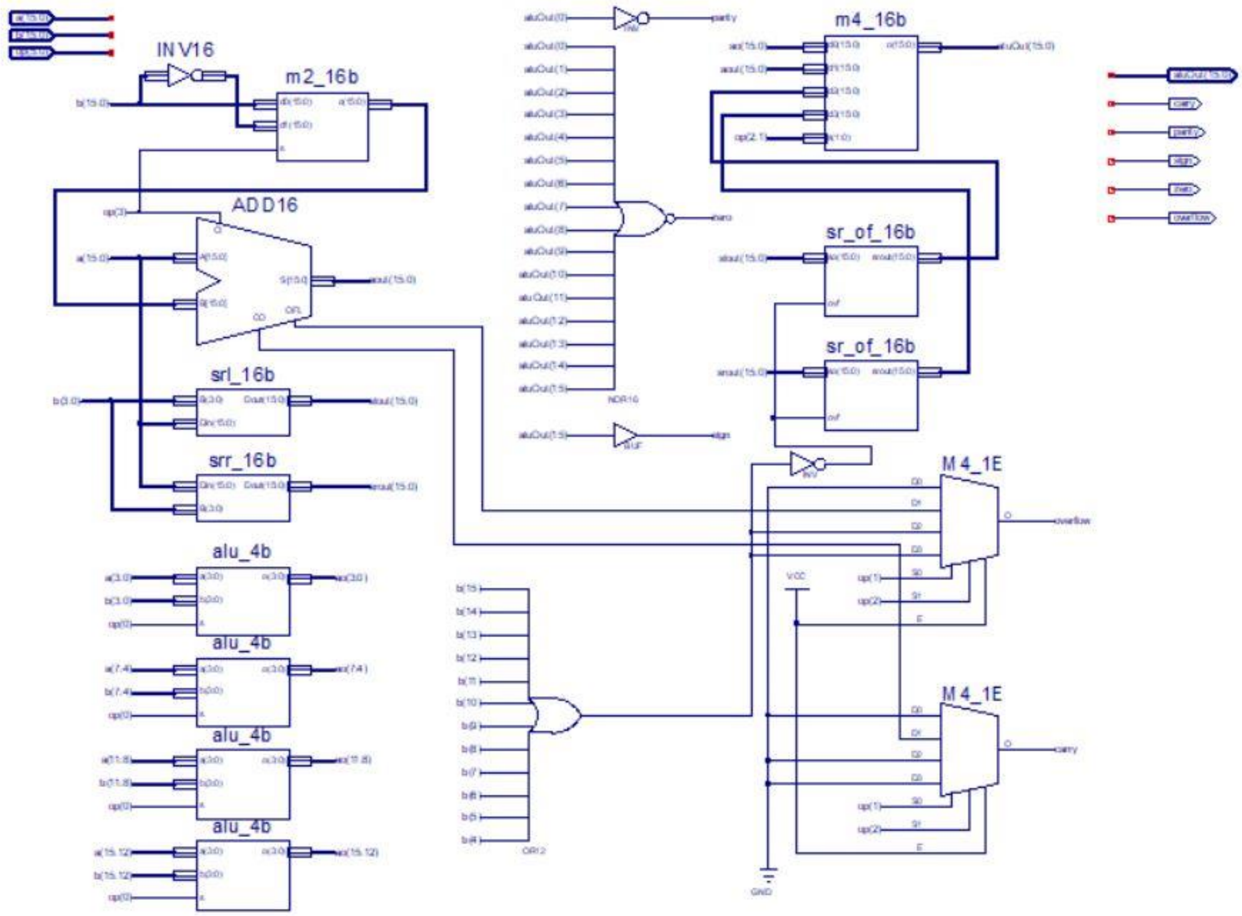

Figure 12-3 ALU

12.2.1.1 OR/AND Operator

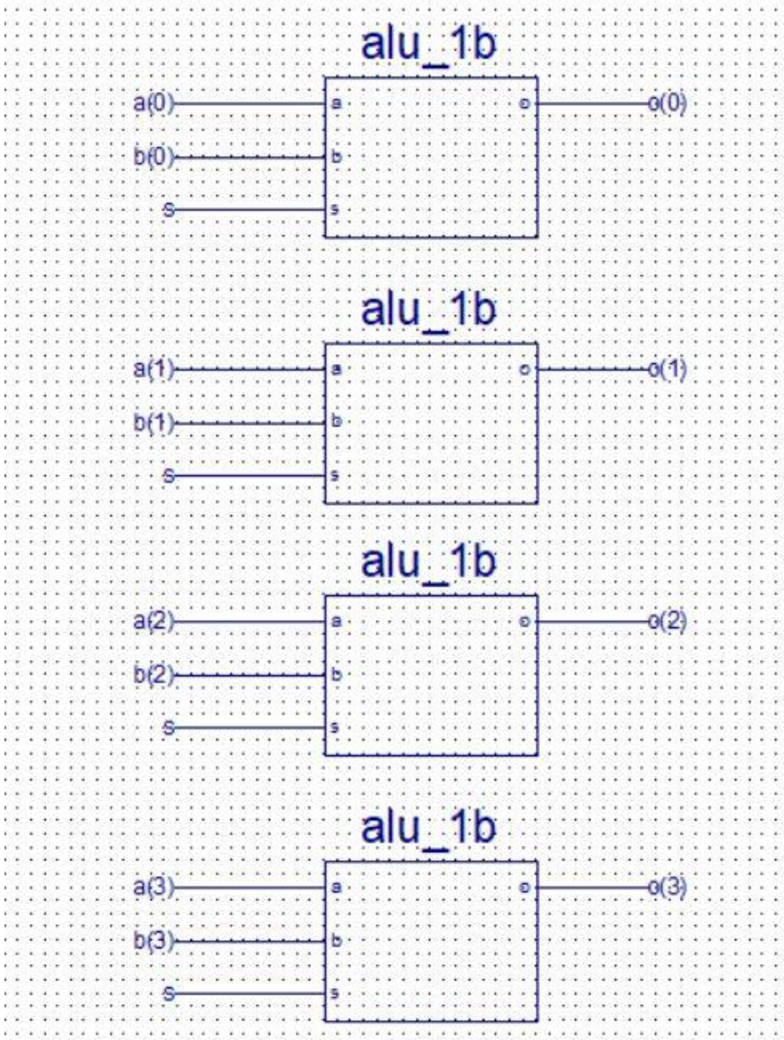

Figure 12-4 ALU OR/AND Operator

12.2.1.1.1 1-Bit AND/OR Operator

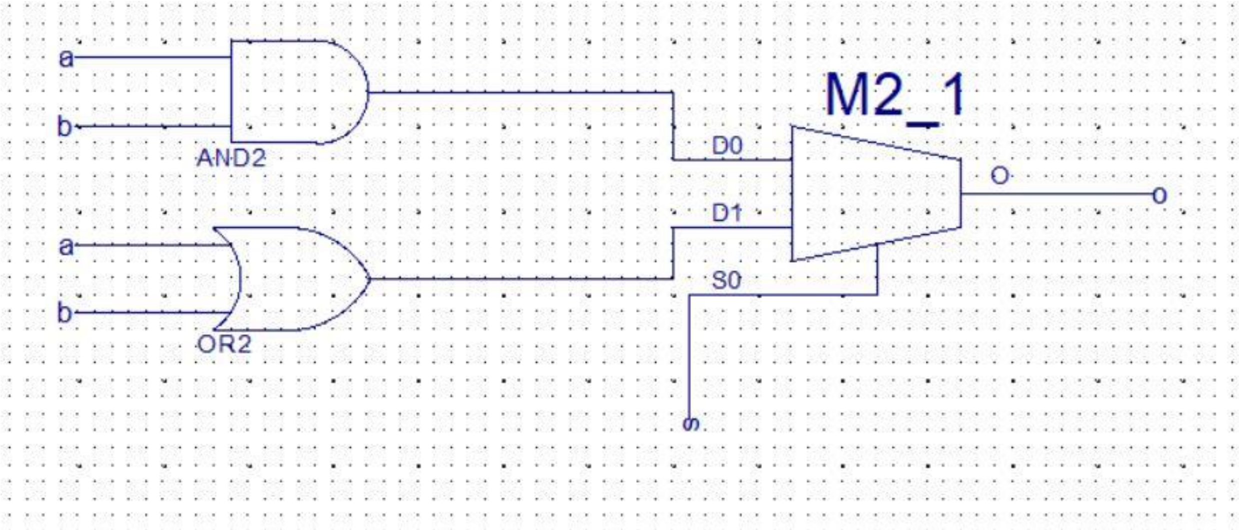

Figure 12-5 1-Bit ALU AND/OR Operator

12.2.1.2 Shifter

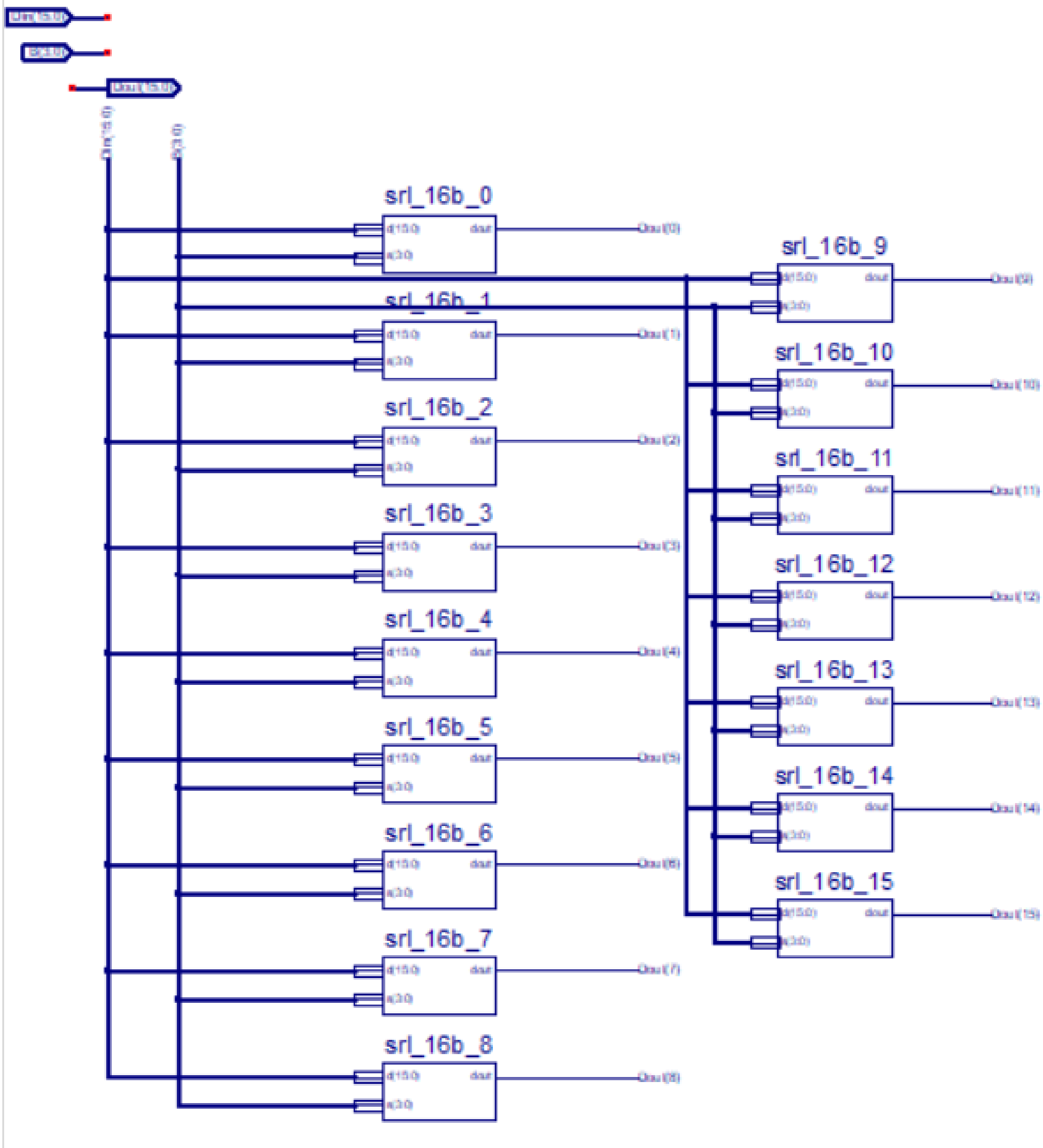

Figure 12-6 Shifter

12.2.1.2.1 One Bit Schematic

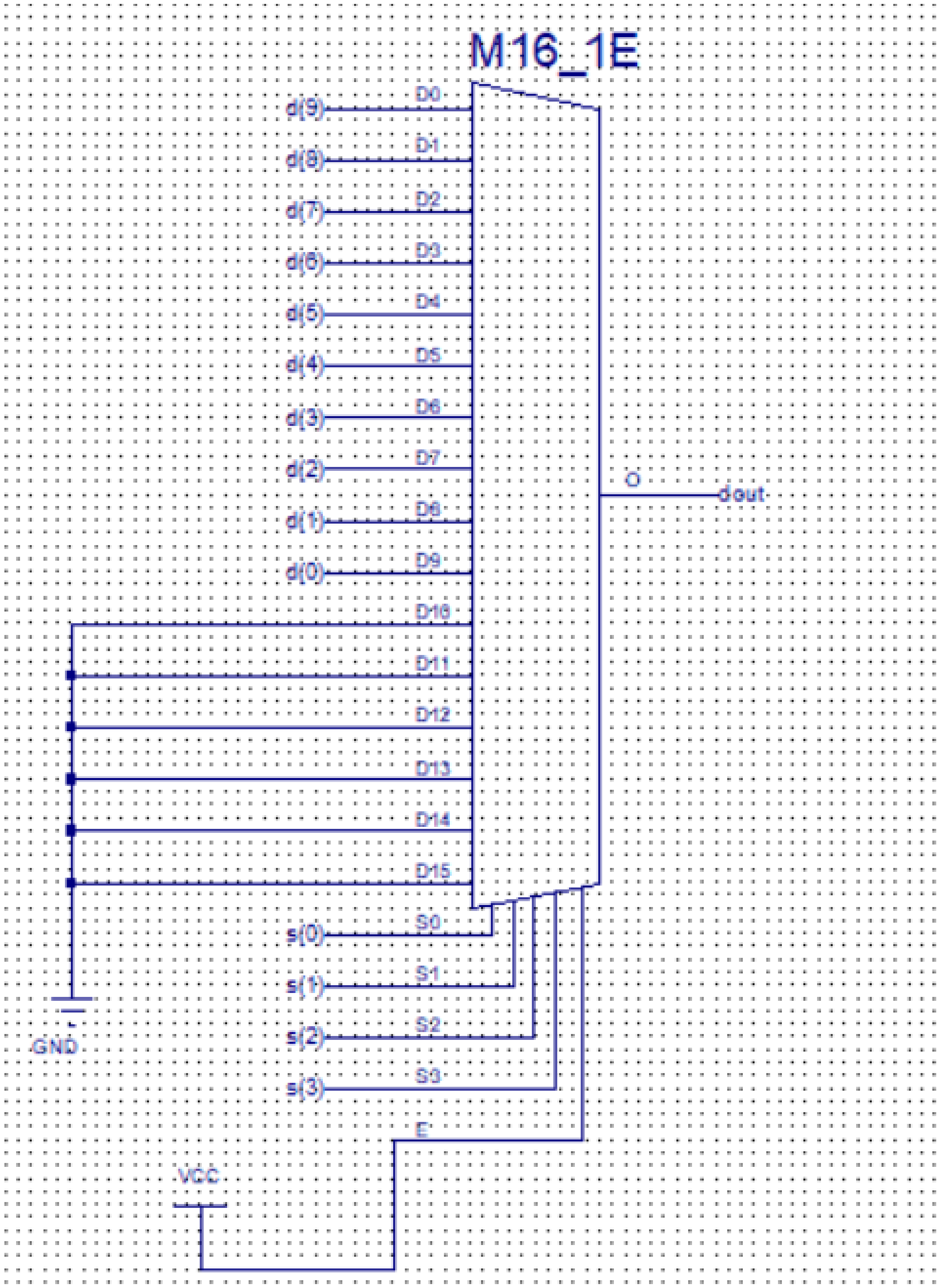

Figure 12-7 One Bit Shifter Schematic

12.2.2 Flag Register

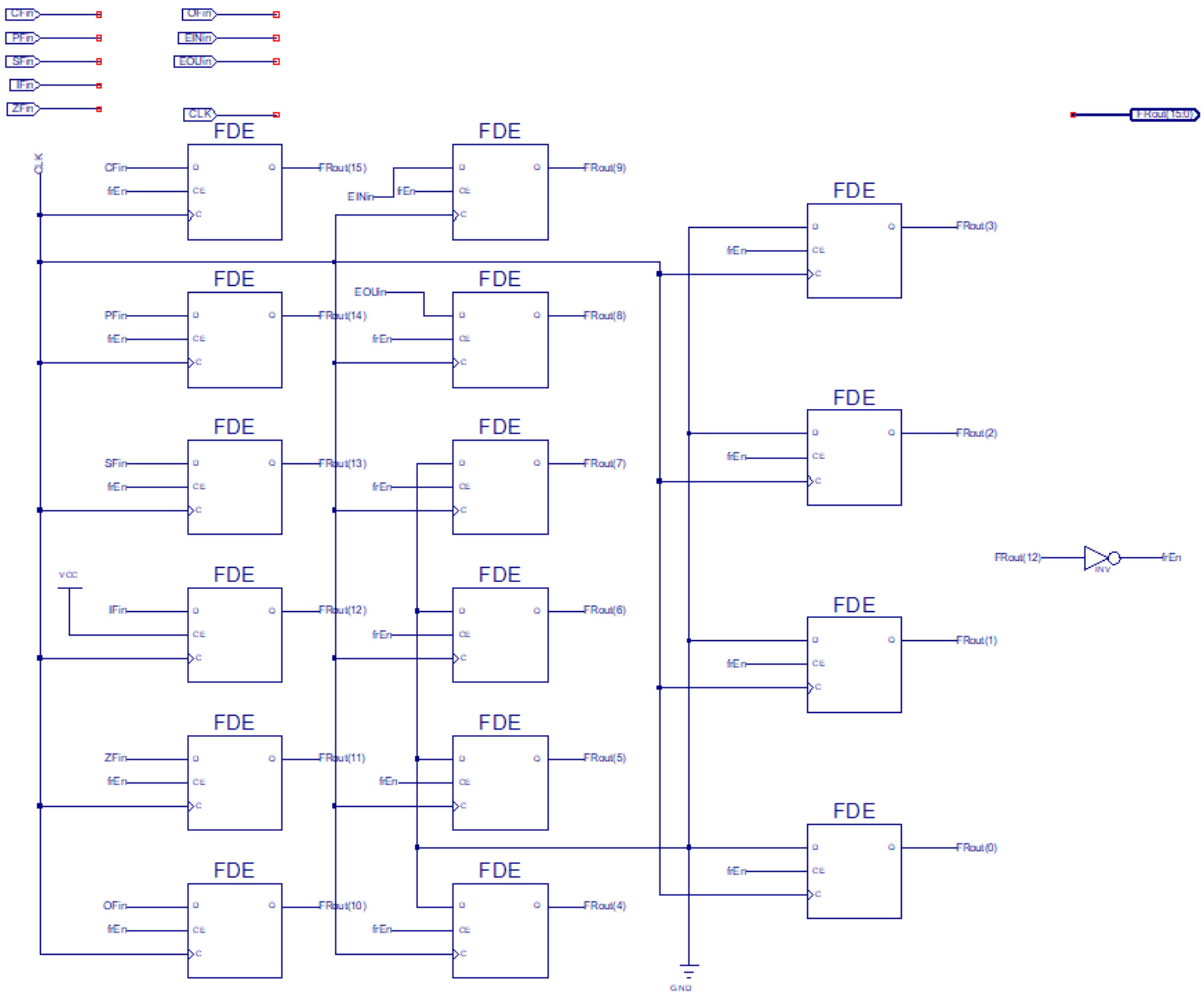

Figure 12-8 Flag Register

12.2.3 Compare State Register

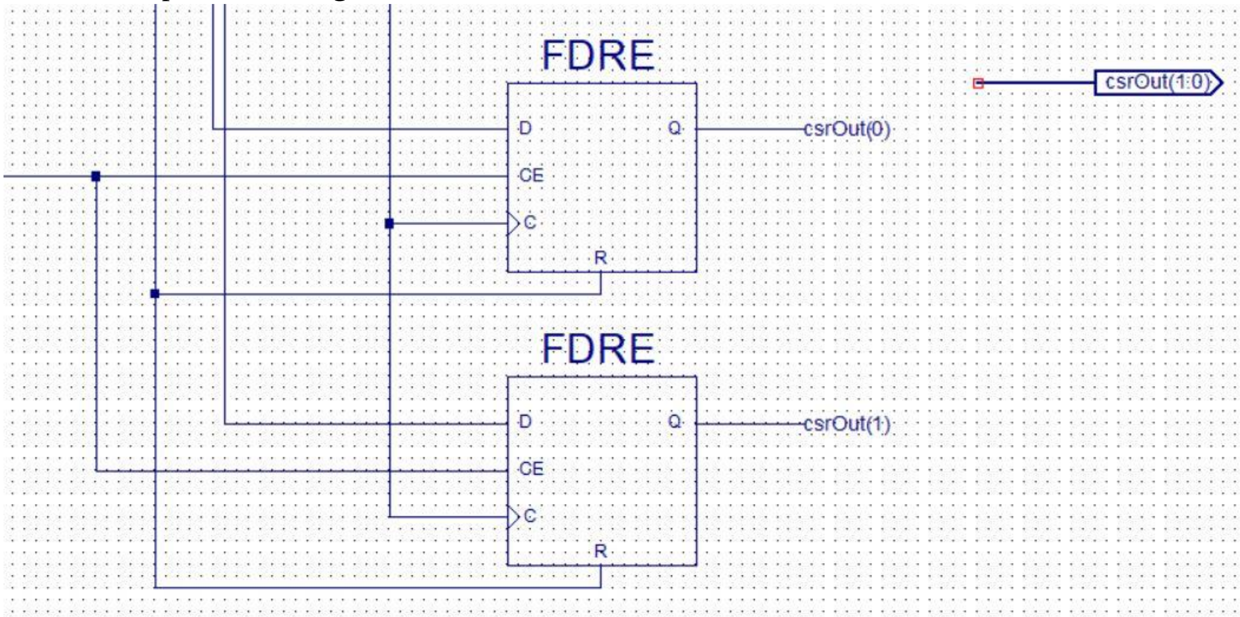

Figure 12-9 Compare State Register

12.3 Subsystem for Fetch Part

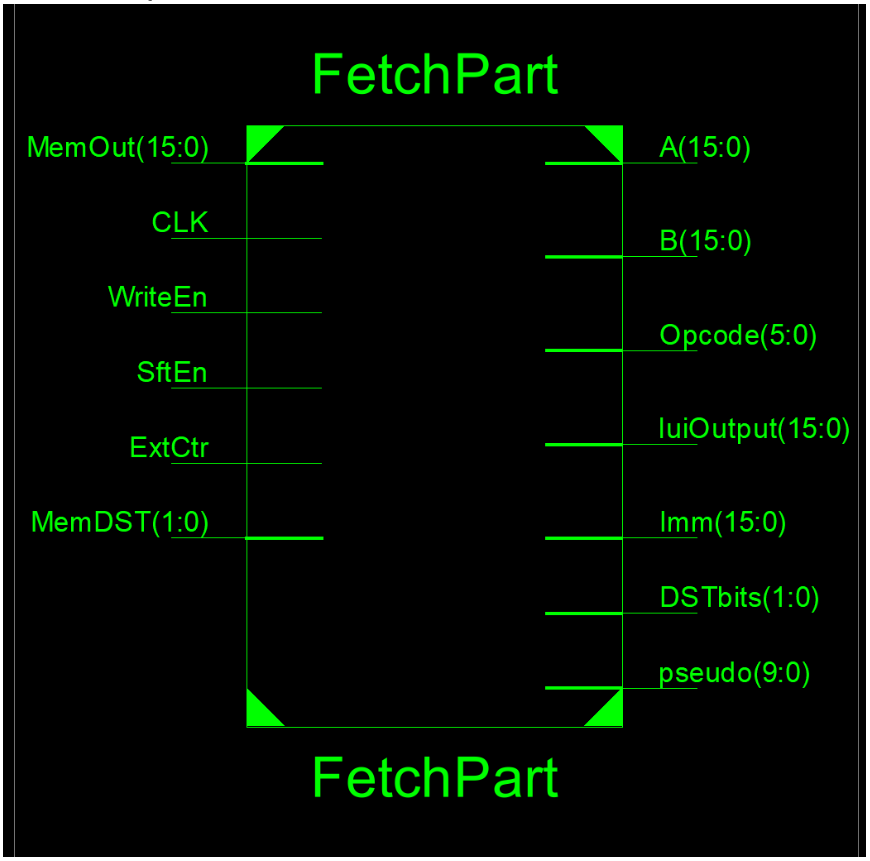

Figure 12-10 Fetch part including shifters, extender and temporary register

12.3.1 Shifter1

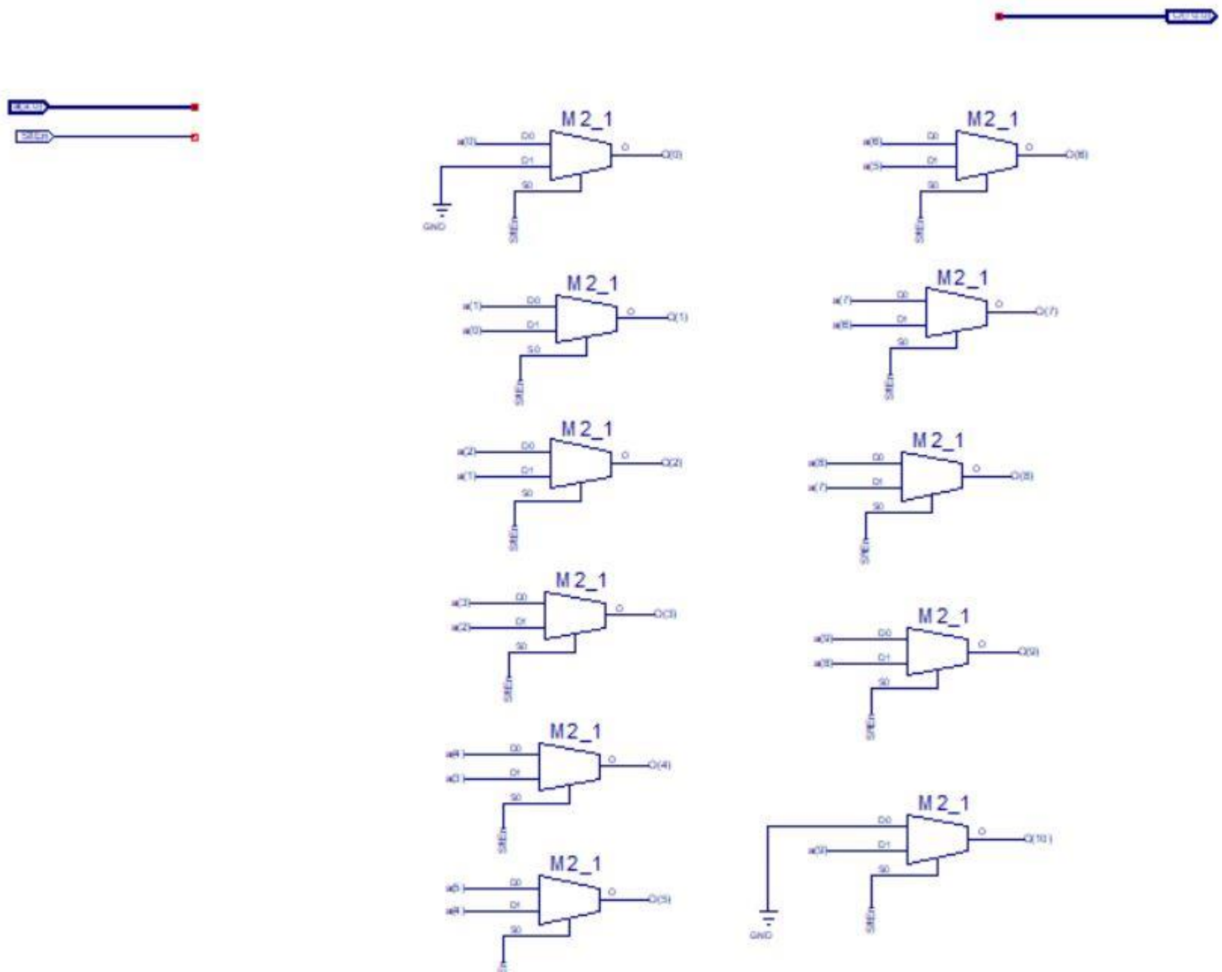

Figure 12-11 1-bit shifter

12.3.2 Shifter10

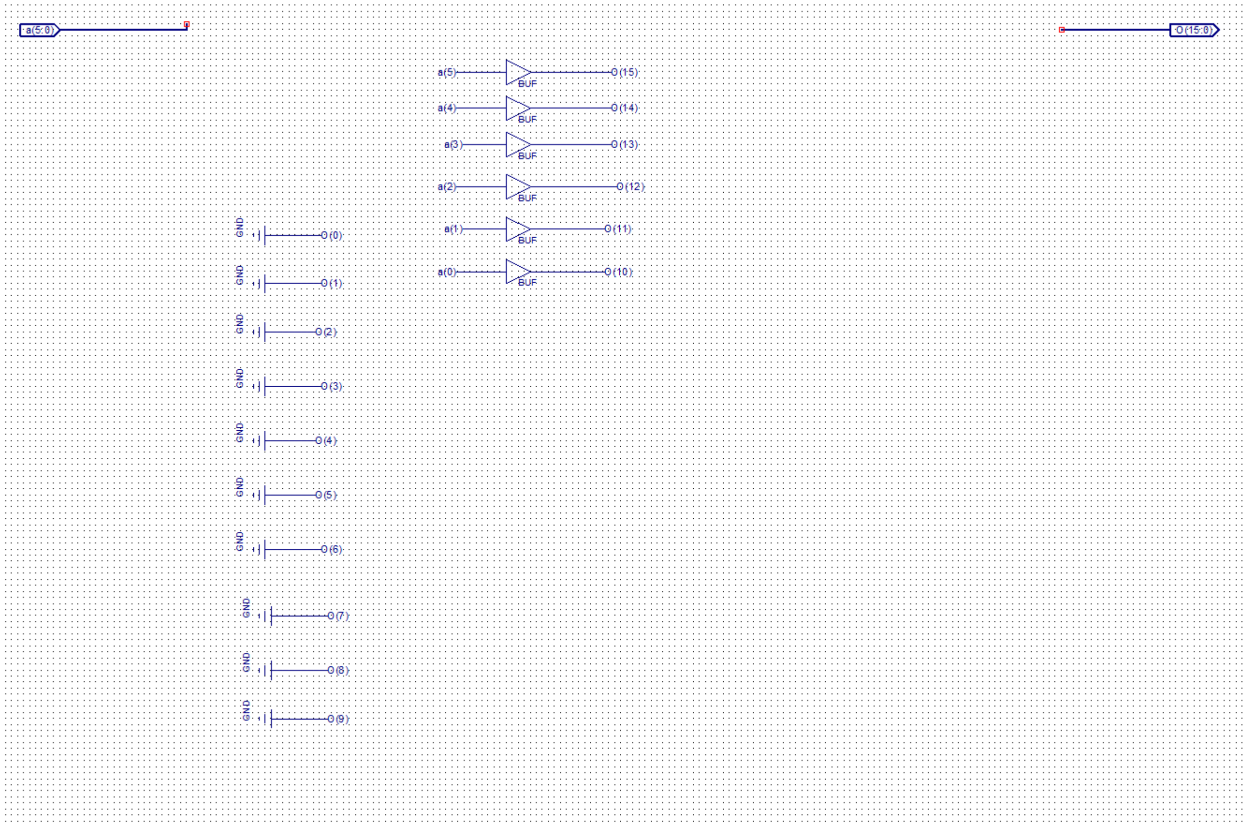

Figure 12-12 10-bit shifter

12.3.3 Extender

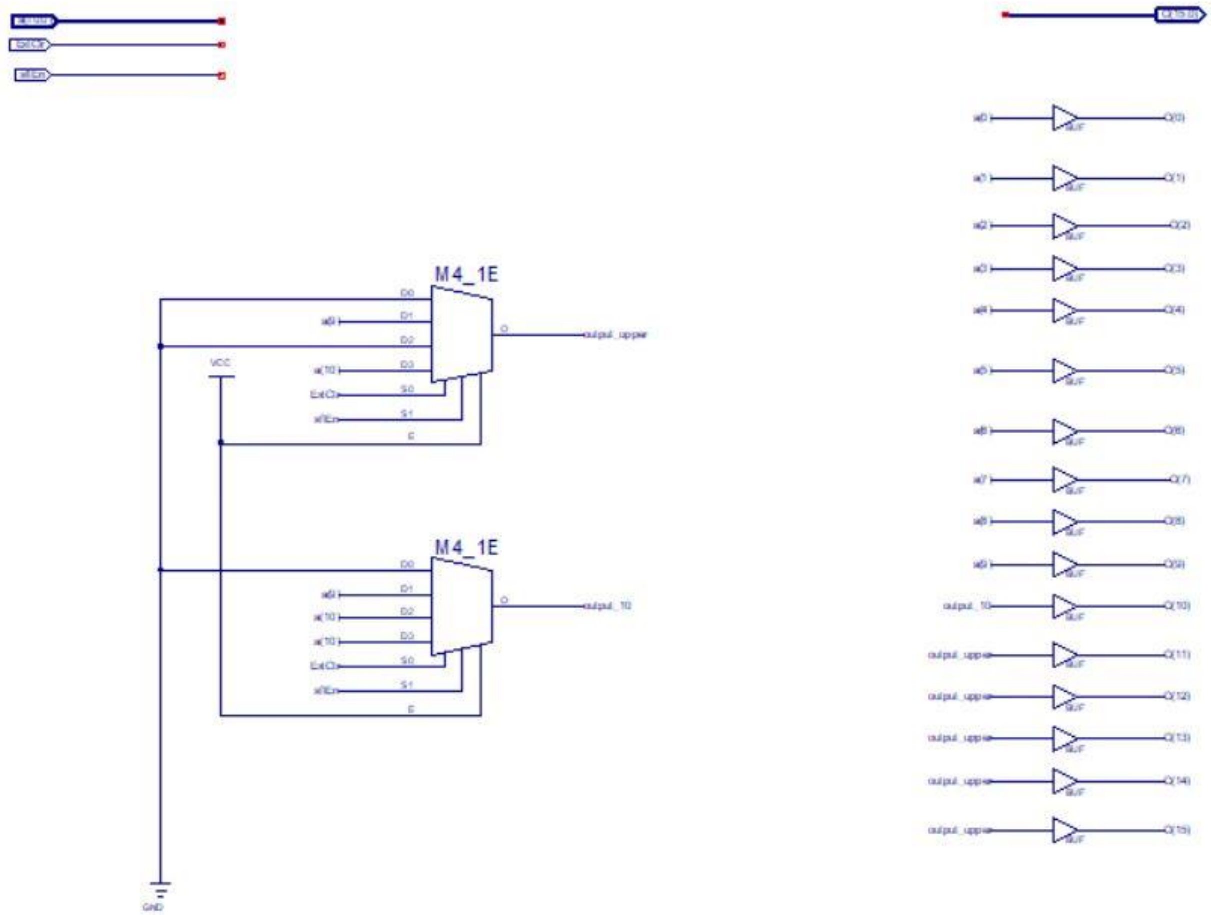

Figure 12-13 Extender

12.4 Subsystem for Stack Pointers

12.4.1 Stack Pointer Registers

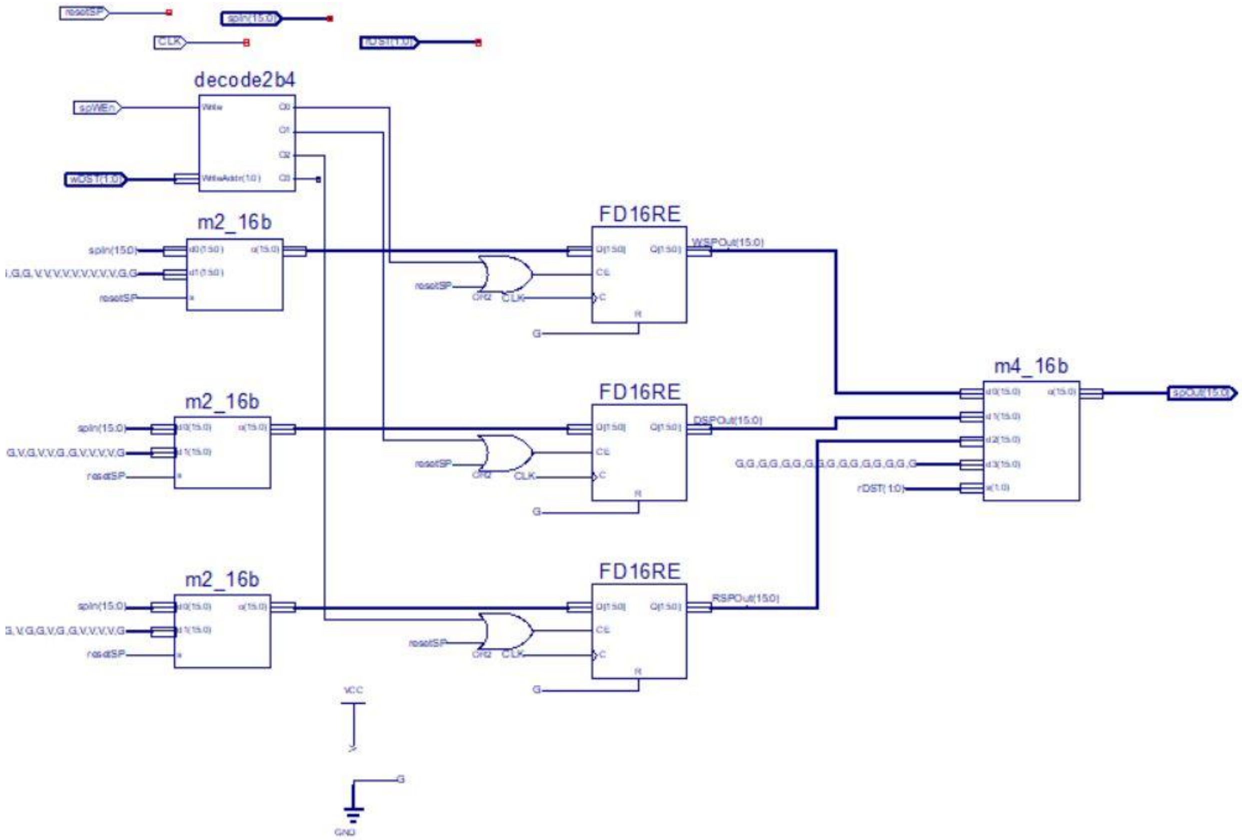

Figure 12-14 Stack Pointer Registers

12.5 Exception Component

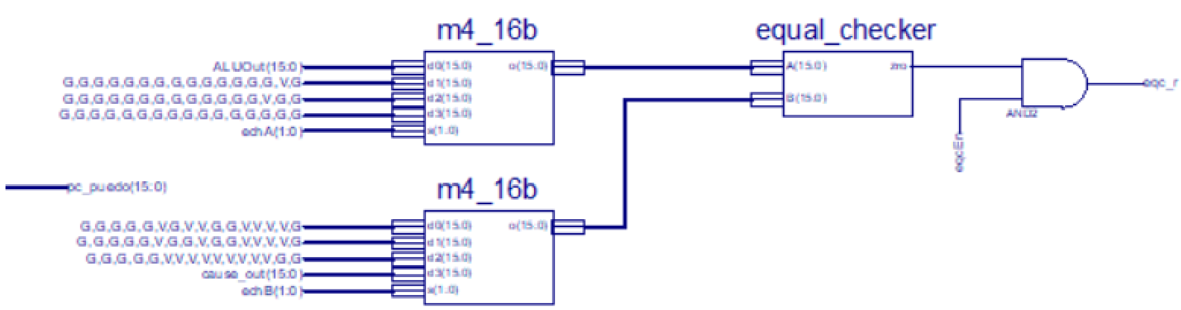

Figure 12-15 Exception Components

12.6 Interrupt Component

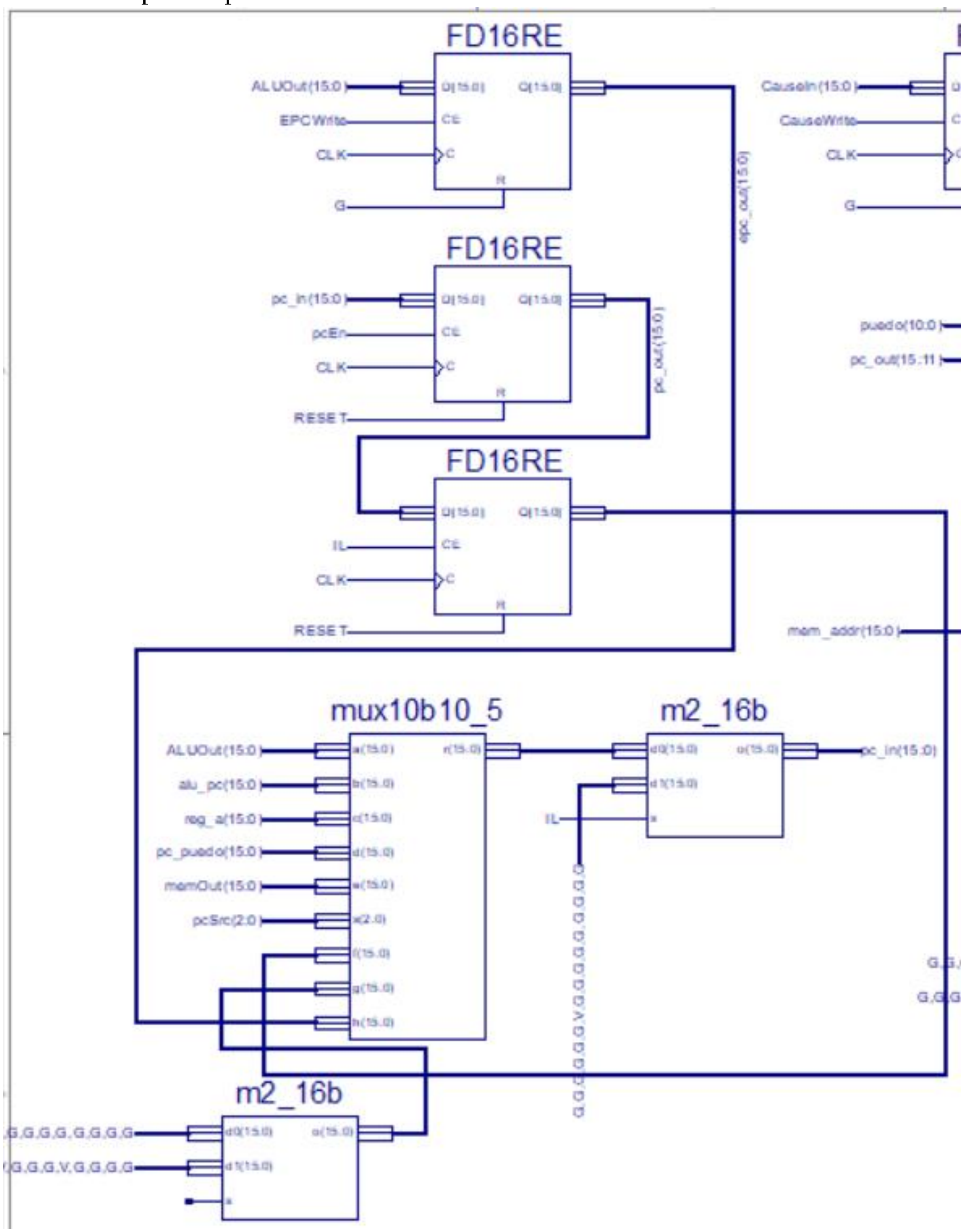

Figure 12-16 Interrupt Components