**Components documentation**

**Common components**

Mux2 nBit

Entity name: mux2\_nbit

Inputs:

* In0 (32 bit) # it’s generic
* In1 (32 bit)
* Sel (1 bit)

Outputs:

* Out0 (32 bits)

nBit\_adder

Entity name: my\_nadder

Inputs:

* a (32 bit) # it’s generic
* b (32 bit)
* Cin (1 bit)

Outputs:

* S (32 bit)
* Cout (1 bits)

Register (in Register file)

Entity name: Reg => Generic n := 32

Inputs:

* Rst (1 bit)
* Clk (1 bit)
* Enable (1 bit)
* RegInput (n-1 bits)

Outputs:

* RegOutput (32 bits)

3x8 Decoder

Entity name: Decoder

Inputs:

* S (3 bits)
* Enable (1 bit)
* oupt (8 bits)

**Decode Stage components**

Register File

Entity name: RegFile

Inputs:

* Rst (1 bit)
* Clk (1 bit)
* WriteAddress (3 bits)
* ReadAddress1 (3 bits)
* ReadAddress2 (3 bits)
* WriteData (32 bits)
* readEnable (1 bit)
* writeEnable (1 bit)

Outputs:

* ReadData1 (32 bits)
* ReadData2 (32 bits)

Decoding Stage

Entity name: DecodeStage

Inputs:

* instruction (32 bits)
* Rst (1 bit)
* Clk (1 bit)
* WriteData (32 bits)
* readEnable (1 bit)
* writeEnable (1 bit)

Outputs:

* ImmValue (32 bits)
* readData1 (32 bits)
* readData2 (32 bits)
* dstAddress (3 bits)
* jumpControlSignals (3 bits) (00: JMP, 01: JZ, 10: JN, 11: JC)
* ALUcontrolSignals (3 bits) (As discussed in ALU component)
* exSrc (1 bit) immediate value bit
* LoadStoreControlSignals (3 bits) (00: LDM, 01: LDD, 10: STD)
* writeBackSignal (2 bits) (00: No WB, 10: WB\_ALU, 11: WB\_MEM)
* MemoryWriteReadSignal (1 bit) (0 for write, 1 for read)
* SPcontrolSignals (3 bits) (00: No change, 01: +1 for POP and RET, 10: -1 for PUSH and CALL)

**Execute Stage components**

ExecuteStage

Description: this component should be connected to the ID/EX buf as input to it and EX/Mem buf is the ouput.

* ALU instructions
  + NOT “000”
  + INC “001”
  + ADD “010”
  + SUB “011”
  + AND “100”

Entity name: ExecuteStage

Inputs:

* Rst, Clk (1 bit)
* ExeSrc (1 bit) # control sig to select Rsrc2 or Immediate
* SETC (1 bit) # control sig to set the carry flag
* AluOpCode (3 bits) # described above in ALU instructions
* Rsrc1,Rsrc2, Immediate (32 bits)
* PCin (1 bit) # PC as input to be changed in jumps

Outputs:

* F (32 bits) # alu result
* PCout (1 bit)

ALU

Description:

* ALU instructions
  + NOT “000”
  + INC “001”
  + ADD “010”
  + SUB “011”
  + AND “100”

Entity name: ALU

Inputs:

* A,B (32 bits)
* opCode (3 bits)
* CCR (3 bits)

Outputs:

* F (32 bits)
* Cout (1 bit)

Flag Control

Entity name: flagControl

Inputs:

* aluRes (32 bits)
* Carry (1 bits) #from ALU
* Setc (1 bits) #control signal to set carry

Outputs:

* CCR (3 bits) #update CCR reg

**Write-Back Stage components**

Write Back Stage

Entity Name: WB\_Stage

Inputs:

* Rst (1 bit)
* Clk (1 bit)
* writeBackSignal (2 bits) #Control signal that is generated at decode
* Memory\_Output (32 bits)
* ALU\_Output (32 bits)
* writeAddressIn (3 bits)

Outputs:

* writeData (32 bits)
* writeAddressOut (3 bits)
* writeEnable (1 bit)

**Memory Stage Components**

Memory

Entity Name: memory

Inputs:

* Clk (1 bit)
* readEnable (1 bit)
* writeEnable (1 bit)
* dataIn (32 bits)
* address (20 bits)

Outputs:

* dataOut (32 bits)

Instructions: Memory Location 0x00000 to Memory Location 0x55555

Data: Memory Location 0x55556 to Memory Location 0xAAAAB

Stack: Memory Location 0xAAAAC to Memory Location 0xFFFFF

Memory Stage

Entity Name: memoryStage

Inputs:

* Clk (1 bit)
* Rst (1 bit)
* CCR (3 bits)

Outputs:

**General Notes**

* Instruction is 32 bits
* Opcode is 5 bits
* 8 registers (R0 -> R7)
* 2 special registers: PC + SP
* Each register is 32 bits
* Flag register (CCR) (3 bits)
  + Z (Zero flag)
  + N (Negative flag)
  + C (Carry flag)
* Control Signals:
  + Write Back Signal (whether we write back or not) (0: No WB, 1: WB)
  + Write Back from ALU or Memory (0: ALU, 1: Memory)
  + Branch (whether there is branch or not) (0: No branch, 1: Branch)
  + Type of branch (00: JMP, 01: JZ, 10: JN, 11: JC)
  + Execute 2nd operand from register or from immediate value (0: register, 1: immediate value)
  + Memory Write/Read (0 for write, 1 for read), (all instructions that don’t use memory will have a read signal)
  + SP Control:
    - Control unit outputs a signal that has 3 combinations:
      * 00: No change (false enable)
      * 01: +1 for POP and RET
      * 10: -1 for PUSH and CALL
* Forwarding Unit:
  + Inputs:
    - Write Address from ALU Buffer
    - Write Address from Memory Buffer
    - Current Read Address (Rsrc1 and Rsrc2) from Instruction Fetch
    - //should be address in buffer after decode since if second instruction is waiting for data in alu, it has to wait tell it reaches buffer after alu and in this case the second instruction will have reached buffer after decode
  + Output:
    - Selection line of MUX before Decode Buffers
      * Most significant 2 bits for data 1 and least significant 2 bits for data 2
      * 00 -> From register file
      * 01 -> From ALU
      * 10 -> From Memory
      * Ex:
        + 0000: Read Data 1 from register and Read Data 2 from register file
        + 0001: Read Data 1 from register file and Read Data 2 from ALU
        + 1010: Read Data 1 from Memory and Read Data 2 from Memory
        + 0100: Read Data 1 from ALU and read data 2 from register file
        + 1001: Read Data 1 from Memory and read Data 2 from ALU
  + Logic:
    - Compare Read Addresses 1 & 2 with Write Address from ALU and from memory Buffers

Alu:

* Input:
  + First register address which is the source register from ID/EX buffer
  + second input is out of mux
    - the selection of the mux depends on the instruction
    - if instruction is R-type or branch then the output of the mux is the second register read in decode stage
    - if the instruction is load/store or any instruction using immediate value then the sign extended immediate value is the output of mux
* output:
  + In R-type instruction, the output is the value that will be stored in the destination register and which will also be used in forwarding if needed
  + In load/store instruction the output is the address that will be used in memory stage to load or store
  + In branch instruction the output is the result of comparison which tells us whether to take the branch or not
  + Static Branch Prediction
    - Always not taken to minimize error.
    - Execute successor instruction in sequence.
    - Discard (flush) instructions in pipeline if branch was actually taken.