**Computer Architecture**

**Phase 2 Delivery**

**Design/Instructions/Operations**

**Group: 4**

**Team Members:**

* Ahmed Salah El-Dien Mohamed (1142320)
* Ahmed Maher Abo-Bakr (1142036)
* Omar Alaa Eldin (11417003)
* Karim Ashraf (1142376)

Instruction Format:

**FInalized Groups & Func Bits:**

The instructions (needed for the assembler) (each instruction is 16 bits due to memory width (except LDM), so wasted bits in some instructions are wasted because they need less than 16 bits. For parsing the instructions, the instruction in itself is 3 letters (so switch case) and then depending on instruction take the arguments. If single argument, take the rest of the line. If two arguments, take value till comma then till end of line.

* 1. (NULL-Type)  
     Bit-format (000 0000000000 XXX) where the XXXs are indicated as follows:   
     NOP<000>, SETC<001>, CLRC<010>, RET<011>, RTI<100>
  2. (R-Type)   
     Bit format (001 SSS DDD 0000 XXX) Ss for Rsrc bits Ds for Rdst bits and Xs indicated as follows:   
     MOV<000>, ADD<001>, SUB<010>, AND<011>, OR<100>
  3. (A-Type)  
     Bit-format (010 SSS DDD 000 XXXX) (Duplicate R in instruction in SSS and DDD  
     RLC<0000>, RRC<0001>, PUSH<0010>, POP<0011>, OUT<0100>, IN<0101>, NOT<0110>, NEG<0111>, INC<1000>, DEC<1001>
  4. (S-Type)  
     Bit-format (011 SSS DDD VVVVV 0 X) Vs are immediate value and Xs as follows:  
      SHR<0>, SHL<1>
  5. (I-Type) (depending on V put Z so as to make upper 16 bits != lower 16 bits)  
     Bit-format (100 SSS SSS 000000Z - VVVVVVVVVVVVVVVV)  
     LDM
  6. (X-Type)  
     Bit-format (101 SSS EEEEEEEEE X) Es are for effective address value  
     LDD<0>, STD<1>
  7. (J-Type)  
     Bit-format (110 SSS 000000 XXX)  
     JMP<000>, JZ<001>, JN<010>, JC<011>, CALL<100>

**Groups’ Opcodes:**

1. Null-Type: <000>
2. R-Type: <001>
3. A-Type: <010>
4. S-Type: <011>
5. I-Type: <100>
6. X-Type: <101>
7. J-Type: <110>

**Control Bits: (Some control bits were edited or added)**

1. **Fetch-related signals for instruction**: PCMux (Normal adder, Decode R1, Ex R1, Memory)=>each put in own stage flags not actually in fetch stage, FetchMux(instruction, PC), PCEnable(default 1)
2. **Decode:** MuxImm(lower 16, 9, 5) //Immediate-Effective address-Shift value
   1. All operands are 16 bits by adding ground (0s) to ‘9’ & ‘5’
3. **Execute:** Op1 MUX (R1,,,SP, Imm value) , Op2 MUX (R2,,,Immediate) , ALU Op\*, EnWriteFlagReg, Carry MUX (Zeros, Flags), SPWriteEn, MUX-MEMSrc (ALU, Operand1)
   1. **ALU Op bits (ordered in increasing bit representation with 4 bits):** OR<0000>, AND, ADD, SUB, RLC, RRC, SHL, SHR, SETC, CLRC, INCOperand1, DECOperand1, NOT, NEG<1110>
4. **Memory bits:** MemWrite (1 enable), MemRead (1 enable), OUTPortWrite (1 enable), MemAddressMux(address from Execute, zeros)
5. **Write back:** RegWrite (1 enable), MuxMemToReg (alu, memory,out)

**Finalized Processor Logic: (need to inspect new diagram for reference)**

1. Branch Decision Unit:
   1. If (Jmp indictor = 1 & Zero flag = 1 || JI = 2 & N = 1 || JI = 3 & C = 1)
      1. Force jmp = 1
   2. Else
      1. Force jmp = 0
2. Control unit (has a single backward counter & a 2 bit register)
   1. 2 Bit Register:
      1. If counter == 0: reset bit
      2. If null type & rti: value = 1
   2. Backward counter:
      1. If NULL TYPE:
         1. If RET or RTI: start the backward counter from 3
   3. Jmp Indicator
      1. If J-type: put lower two bits into the Jmp indicator buffer on rising edge of clock (this means Call and JMP are zero)
   4. Buffers control : (Flush means make the input of the buffer zeros so as to take zeros in the next rising edge of the clock)
      1. If Force JMP: flush fetch & decode (overrides next if condition)
      2. If J-type:
         1. If JMP: flush fetch buffer
         2. If call & upper 16 bits are the same instruction bits: stall lower 16 bits of buffer (we will have as if to CALL cycles to prepare the full instruction, so we need to have info if this is the first iteration or the 2nd. This indicator will be if PC is present in the lower bits or not. The PC will always != CALL instruction because the PC’s value is actually 9 bits only (but the PC itself is 16 bits)
      3. If backward counter > 1, flush fetch buffer
      4. If LDM & upper 16 bits are the same instruction bits: stall lower 16 bits
   5. In Mux selector
      1. Always 0 except if A-Type and IN function
   6. PC MUX
      1. If forced JMP: value = 3 (Notice overrides the next if)
      2. If J-type
         1. If JMP or call: value = 0
      3. If counter != 0, value = 1
      4. Default value is 2
   7. Imm Value MUX Selector:
      1. Default don’t care
      2. If S-type, value = 0
      3. If X-type, value = 2
      4. If call & upper 16 bits are not the same instruction bits: value = 1
      5. If LDM & upper 16 bits are not the same instruction bits: value = 1
   8. Upper MUX Selector
      1. Default don’t care
      2. If J-type & call & upper 16 bits are the same instruction bits: value = 1
   9. EX CU bits (AA B CCCC D E F H G L) AA is MSB ?
      1. What they represent:
         1. AA (represent Op1 MUX Selector)
         2. B (represents Op2 MUX Selector)
         3. CCCC (represents ALU Op)
         4. D (represents Enable Write Flag Register)
         5. E (represents Carry MUX selector)(0 is zeros, 1 is carry)
         6. F (represents SP Write Enable)
         7. H (represents Ex Out MUX selector)
         8. G (represents Flags MUX selector)(0 is ALU 1 is memory)
         9. L (represents Flag or Op2 MUX)
      2. Logic:
         1. If register == 1 (overrides other if conditions)
            1. If counter == 2: 10 0 1010 0 0 1 0 0
            2. If counter == 1: 00 0 0000 1 0 0 0 1
         2. If LDM & upper 16 bits are not the same instruction bits:
            1. 01 0 0000 0 0 0 0 1 0
         3. If NULL type:
            1. If setc: 00 0 0110 1 0 0 0 0
            2. If clrc: 00 0 0111 1 0 0 0 0
            3. If ret or rti: 10 0 1010 0 0 1 0 0
         4. If A-type:
            1. If push: 10 0 1011 0 0 1 1 0
            2. If pop: 10 0 1010 0 0 1 0 0
            3. If out: 00 0 0000 0 0 0 0 0
            4. If in: 00 0 0000 0 0 0 1 0
            5. If rlc: 00 0 0100 1 1 0 0 0
            6. If rrc: 00 0 0101 1 1 0 0 0
            7. If not, neg, inc, or dec: 00 0 XXXX 1 0 0 0 0 where Xs are as follows   
               OR<0000>, AND<0001>, ADD<0010>, SUB<0011>, RLC<0100>, RRC<0101>, SHL<0110>, SHR<0111>, SETC<1000>, CLRC<1001>, INCOperand1<1010>, DECOperand1<1011>, NOT<1100>, NEG<1110>
         5. If R-type:
            1. If not move instruction (func != 0)

00 0 XXXX 1 0 0 0 0 where Xs are as follows   
OR<0000>, AND<0001>, ADD<0010>, SUB<0011>, RLC<0100>, RRC<0101>, SHL<0110>, SHR<0111>, SETC<1000>, CLRC<1001>, INCOperand1<1010>, DECOperand1<1011>, NOT<1100>, NEG<1110>

* + - * 1. If move instruction:

00 0 0000 0 0 0 1 0

* + - 1. If S-type:
         1. 00 1 XXXX 0 0 0 0 0 where Xs are as follows   
            OR<0000>, AND<0001>, ADD<0010>, SUB<0011>, RLC<0100>, RRC<0101>, SHL<0110>, SHR<0111>, SETC<1000>, CLRC<1001>, INCOperand1<1010>, DECOperand1<1011>, NOT<1100>, NEG<1110>
      2. If X-type: 10 0 0000 0 0 0 1 0
      3. If J-type:
         1. If call & upper 16 bits are not the same instruction bits:

10 1 1011 0 0 1 1 0

* 1. M CU bits (A B C)
     1. What they represent
        1. A for Mem Write Enable
        2. B for Mem Read Enable
        3. C For Out port write enable
     2. Logic
        1. If register == 1 (overrides other if conditions)
           1. If counter == 2: 0 1 0
           2. If counter == 1: 0 0 0
        2. If LDM & upper 16 bits are not the same instruction bits:
           1. 0 0 0
        3. If Null type:
           1. If ret or rti: 0 1 0
        4. If J-type & call & upper 16 bits are the same instruction bits:
           1. 1 0 0
        5. If A-Type
           1. If push : 1 0 0
           2. If pop: 0 1 0
           3. If rlc,rrc,in,not,neg,inc,dec: 0 0 0
           4. If out: 0 0 1
        6. If R-type or S-type: 0 0 0
        7. If X-type:
           1. If LDD, 0 1 0
           2. If STD, 1 0 0
  2. WB CU bits (A B)
     1. What they represent
        1. A for Reg Write Back Enable
        2. B for Value to Reg MUX selector
     2. Logic
        1. If A-type:
           1. If rlc, rrc, in, not, neg, inc, dec: 1 1
           2. If pop: 1 0
           3. If out, push: 0 0
        2. If R-type or S-type: 1 0
        3. If LDM & upper 16 bits are not the same instruction bits:
           1. 1 1
        4. If X-type
           1. If LDD, 1 0
           2. If STD, 0 0

1. Forwarding unit
2. Hazard detection unit
3. Fetch stage
   1. PC MUX selects to output depending on selector. All inputs to the MUX are inputs from other modules except the PC+1 input, so make an adder.
   2. PC (if enabled), acts as a dummy 16 flip flop on rising edge of clock.
   3. Instruction memory is normal memory that outputs whatever the address is on its input port, no need to even look at the clock; no write will be made.
   4. Lower MUX acts like a normal MUX and its selector is an input to this module
   5. For the fetch buffer, the upper 16 bits take the output of the instruction memory each rising edge of the clock while the lower 16 bits take the output of the lower MUX each rising edge
4. Decode stage
   1. Two values need to concatenate zeros to them as indicated in the diagram with bits specified
   2. There are 4 MUXs that have selectors as input to this module (R1 & R2 MUXS are for choosing forwarded value or not, Imm value MUX for choosing what is the immediate value bits since they vary in the instruction, and the other for treating the IN port as just a register to make the IN instruction look like a MOV instruction)
   3. There is a register file that on rising edge write the value on the write port to the write address if write enable is 1 and on falling edge always decodes R1 and R2 addresses and outputs them on its data out ports respectively.
   4. The decode buffer is 48 bits aside from the buffered control bits. Whether you choose to merge the decode buffer with the buffered control bits is your choice, in this document I will consider them separate. Of course as previously stated, the buffer acts as dummy flip flop that reads on rising edge of clk
   5. The 48 bits are divided as follows: Value from IN MUX (16 bits), Value from R2 MUX (16 bits), Immediate Value 16 bits
   6. Note that MUX R2 outputs its value from this module to the previous one
5. Execute stage
   1. There are four MUXs with selectors as input to this module
   2. The SP is a register that will take the value from the ALU on the rising edge if the write enable = 1
   3. The ALU performs whatever operation the ALU Op bits specify on Op1 and Op2. Of course, the ALU also outputs flags. These flags are written to the flag register on the rising edge if the enable write flag is 1 except for the following case. In case of RTI, we need to write on the flag register from the memory, so we put a Flags MUX that normally chooses flags from the ALU or the MEM in case of RTI.
   4. On the rising edge of the clock, the Ex buffer is updated with the following values. The first 16 bits take the Mem address from the bits of the Ex Out MUX, the 2nd 16 bits take the Mem write value and the last 3 bits take R2 as a write back address for the write back stage. The M and WB CU bits transferred from the decode stage are assumed not to be part of this buffer but you can assume otherwise.
6. Memory stage
   1. There is a single MUX that will pass a hardcoded address, zero, if there is a reset signal.
   2. The out ports outs the write value if the port is enabled
   3. The data memory outputs data or write data on it depending on what is enabled with the rising edge of the clk, both cannot happen at the same time.
   4. In the Mem buffer, the first 16 bits are the Mem Value, the 2nd 16 bits are the Write Value (basically we are choosing ALU or Mem to write back to register), and the last 3 bits are the WB Register address.
   5. Again, the WB CU Bits are transfered as is and are assumed as not part of this buffer
7. Write back stage
   1. This is too clear from the diagram

Hazards:  
1. Structural hazards are resolved by writing in rising edge and reading in falling edge in register file

2. Control hazards are resolved by the above instructions guide

3. Data hazards are resolved by a forwarding unit that assumes max stage time is execute and not memory; it goes as follows:  
If in the decode instruction there is a register that will be written back on from Ex or Mem stages and its data is needed (notice not all instructions that have R2 need the value of R2 so it depends):  
 If the instruction is in Ex and will read Mem: then hazard detection will flush (notice CALL overrides that rule)  
 Else forward as shown in diagram (IN is not resolved, and we need to change the place of IN)  
It is worth noting that we need to override Call’s signal in load use case and to stall twice if we are going to write from IN port