UNIVERSITY OF CALIFORNIA, DAVIS
Department of Electrical and Computer Engineering

EEC180B **Lab 7: MISP Processor Design** **Spring 1995**

Objective: In this lab, you will complete the design of the MISP processor, making use of your work from the register file design, ALU design, and memory interface design labs done previously. Using the PowerView CAD tools, you will enter your design as a schematic and verify it through functional simulation. Then, using the XACT software, you will implement your design in a Xilinx 4005 FPGA and download it to a demo board for testing and verification.

I. Instruction Set

The MISP processor has an instruction set consisting of only 11 instructions. There are three memory access instructions (load immediate, load and store), four ALU instructions (add, increment, negate and subtract), three program control instructions (jump, branch if zero and branch if negative) and the nop instruction. The instruction formats are shown in Figure 1 below.

Instruction	Mnemonic	Opcode	Operands		Function
no operation	NOP	0000	xx	xx	
load immediate	LDI Rd	0001	Rd	xx	Rd ← M[IAR+1]
load	LD Rd, Ra	0010	Rd	Ra	Rd ← M[Ra]
store	ST Rd, Ra	0011	Rd	Ra	M[Ra] ← Rd
add	ADD Rd,Ra	0100	Rd	Ra	Rd ← Rd + Ra
increment	INC Rd	0101	Rd	xx	Rd ← Rd + 1
negate	NEG Rd,Ra	0110	Rd	Ra	Rd ← - Ra
subtract	SUB Rd,Ra	0111	Rd	Ra	Rd ← Rd - Ra
jump	JMP Ra	1000	xx	Ra	IAR ← Ra
branch if zero	BRZ Ra	1001	xx	Ra	IAR ← Ra if Z=1
branch if negative	BRN Ra	1010	xx	Ra	IAR ← Ra if N=1

Figure 1: MISP Instruction Formats

The upper four bits of the instruction contain the op-code which identifies each specific instruction. The lower four bits are divided into two two-bit operand fields which are each used to specify a register in the register file. Operand fields which contain "xx" are considered "don't cares", since those bits are not used by the instruction.

II. Instruction Operations

The register transfer operations shown in Figure 2 are similar to those presented in the memory interface design lab, with the addition of three new instructions: JMP, BRZ and ADD. The BRZ instruction operations are essentially identical to the BRN instruction except that a different status flag is involved. The ADD instruction register transfer operations also apply to the other ALU instructions as well. (Note that Rd2 refers to the Rd register specified in the instruction, which has been passed through two delay stages).

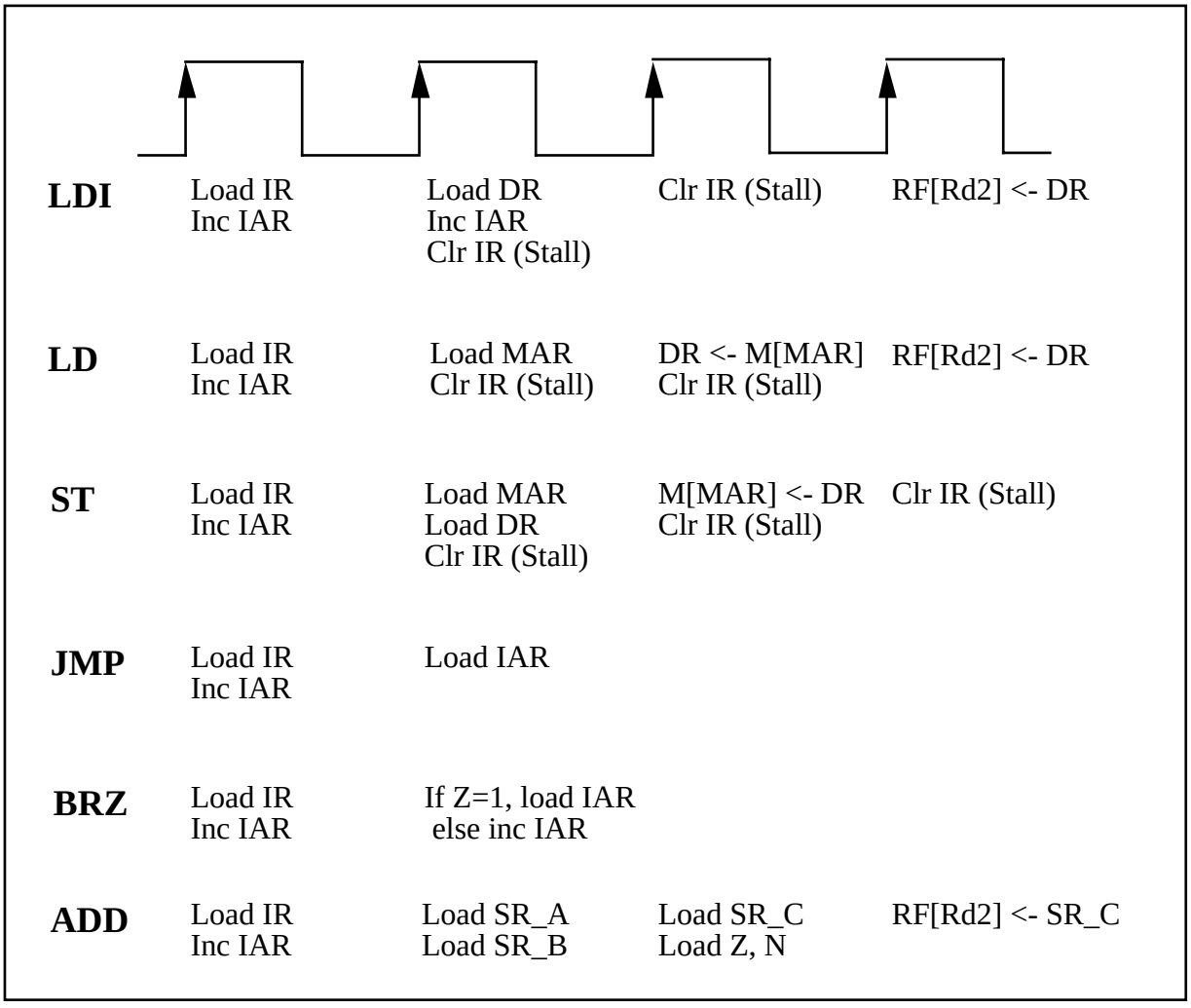

Figure 2: MISP Instruction Operations

In order to understand the MISP instruction operations shown in Figure 2, you should refer to Figure 3, the MISP data path, which shows the interconnections between important registers and

major components such as the register file, ALU, control unit and the external memory. This figure does not show control signals or the actual components to use in your design.

The MISP processor has a four-stage pipeline consisting of the following stages: Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), and Write-Back (WB). The staging registers, SR_A, SR_B, and SR_C, are used to divide the pipeline stages. For an ALU instruction, the operation may be described as follows:

IF: Fetch instruction from EPROM and latch into the Instruction Register (IR).

ID: Decode operand fields from IR and use to address registers in RF. Latch contents into staging registers SR_A and SR_B.

EX: The ALU operates on the contents of SR_A and SR_B. The result is latched into SR_C. For an ALU instruction, the status signals, Z and N, are latched into the status register.

WB: For an ALU instruction, the result in SR_C is written into the appropriate register in RF.

For simplicity of control, the staging registers, SR_A, SR_B and SR_C, should be loaded on every clock cycle. However, the status register will only be updated and the ALU result will only be written into the RF for valid ALU instructions .

For the JMP, BRZ and BRN instructions, the instruction following the jump or branch instruction will always be executed, regardless of whether the branch is taken or not. The instruction following the jump or branch instruction is said to be in the "branch-delay slot". A useful instruction can often be moved from before the branch or jump instruction into the branch delay slot. However, the instruction must not affect the outcome of a conditional branch. If a useful instruction cannot be found for the branch delay slot, then a NOP instruction can be used to insure correct program operation.

As described in the memory interface design lab, most of the cycles of the memory-access instructions cannot be overlapped with other instructions because of the architectural restriction of a common memory address bus and a common memory data bus for both instructions and data. Thus, it is necessary for your control unit to insert "stalls" into the IR during some of the memory-access instruction cycles. The control unit can insert a stall by synchronously clearing the IR. The IAR (instruction address register) should be incremented each time an instruction, including a NOP instruction, is loaded from EPROM or a data byte is read during an LDI instruction. However, it should not be incremented when a stall is inserted into the IR by the control unit.

III. Data Forwarding

The MISP processor has a feature known as data-forwarding. Notice from the MISP data path that the staging registers SR_A and SR_B can be loaded from three different sources: the register file, the ALU output, or the SR_C output. An example will help to illustrate the need for data forwarding. First, we will analyze the execution of the following code fragment cycle by cycle:

```
inc r0          ; increment r0
inc r1          ; increment r1
st r0,r1 ; M[r1] <- r0
```

Cycle	Instruction	Operation
1	inc r0	IF: load IR
2	inc r0	ID: SR_B <- R0
	inc r1	IF: load IR
3	inc r0	EX: SR_C <- R0+1
	inc r1	ID: SR_B <- R1
	st r0,r1	IF: load IR
4	inc r0	WB: RF[R0] <- SR_C
	inc r1	EX: SR_C <- R1+1
	st r0,r1	ID: DR <- R0, MAR <- R1
5	inc r1	WB: RF[R1] <- SR_C
	st r0,r1	EX: M[MAR] <- DR

The need for data-forwarding can be seen in clock cycle 4. Here the MAR and DR registers are latched in the store instruction's ID cycle. However, if these registers are loaded from the register file, then the old values of R0 and R1 will be used, resulting in incorrect operation. In order to use the incremented values of R0 and R1 in the store instruction, we can load DR with the output of SR_C and MAR with the output of the ALU.

In order to implement data-forwarding, the control unit must compare the destination register, Rd, of instructions in the pipeline with the source and destination registers, Ra and Rd, of the current instruction. The rules for data-forwarding in the MISP processor can be summarized as follows:

SR_A <- ALU output if Ra = Rd_1 and op_1 is a valid ALU instruction
else SR_C output if Ra = Rd_2 and op_2 is a valid ALU instruction
else RF[Ra]

SR_B <- ALU output if Rd = Rd_1 and op_1 is a valid ALU instruction
else SR_C output if Rd = Rd_2 and op_2 is a valid ALU instruction
else RF[Rd]

Rd_1 and op_1 refer to the Rd field and op-code of the previous instruction, which is one stage ahead of the current instruction in the pipeline. Rd_2 and op_2 refer to the Rd field and op-code of the instruction which is two stages ahead of the current instruction in the pipeline. The data forwarding logic will be used to generate the MUX control signals for the two MUXes which provide the inputs for SR_A and SR_B. As shown in the MISP data path, these MUXes also provide inputs for the MAR, IAR and DR registers, which provides data forwarding for the load,

store, jump and branch instructions which follow ALU instructions.

Data forwarding could also be implemented to allow a BRZ or BRN instruction immediately following an ALU instruction. However, for simplicity, we will not implement this feature in the MISP. Instead, the programmer will be required to insert a NOP or a non-ALU instruction before a BRZ or BRN instruction.

IV. Practical Hints

- In order to manage your schematic, you may want to use the following three techniques:

    1. Use symbols for large circuit modules such as the ALU, register file, and control unit.

    2. Connect control signals by name, rather than using nets. This can help to make your schematic more readable. Nets within the same symbol which have the same name are logically connected.

    3. Use more than one sheet for your top-level schematic. When your top-level schematic is open in ViewDraw, select FILE LEVEL PUSH SHEET to start a new sheet. The new sheet is like an extension of the sheet from which you pushed. Thus, signals may be connected by name.

- The 7-segment display port should be memory-mapped to location FF as in the memory interface design lab.

V. Lab Requirements

1. Using PowerView and the Xilinx component libraries, complete the design as specified.

2. Create a test schematic with the RAM (ram256x8.1) and EPROM (rom256x8.1) components used in the memory interface design. An example circuit is shown in Figure 4. Run **xfsim** on your test schematic and perform a functional simulation. Use the eprom.dat file found in the /afs/ece/classes/eec180b/lab7 directory. The simulation program, which is included below, calculates the Fibonacci numbers and displays them on the 7-segment displays. Verify that your processor executes the program correctly.

3. When your design has been successfully simulated, run **xmake** on your MISP schematic to create a Xilinx bit file. Download your final design to a demo board and verify that it runs the Fibonacci program correctly.

4. Demonstrate your simulation and working circuit to your TA. Turn in your schematics and the number of Xilinx resources that your design used.

Fibonacci number generator test program

Address	Binary Data	Hex	Instruction	Comment

```
00          00010000    10    -- ldi r0       ; r0 <- 0 (first Fib. number)
01          00000000    00    -- 0
02          00010100    14    -- ldi r1       ; r1 <- 80 (RAM address)
03          10000000    80    -- 80
04          00110001    31    -- st r0,r1     ; M[80] <- 0
05          01010000    50    -- inc r0       ; r0 <- 1 (second Fib. number)
06          01010100    54    -- inc r1       ; r1 <- 81 (RAM address)
07          00110001    31    -- st r0,r1     ; M[81] <- 1
08          00010000    10    -- ldi r0       ; r0 <- 80 (RAM address)
09          10000000    80    -- 80
0a          00010100    14    -- ldi r1       ; r1 <- 81 (RAM address)
0b          10000001    81    -- 81

0c    L     00101000    28    -- ld r2,r0     ; r2 <- lower Fib. number
0d          00101101    2d    -- ld r3,r1     ; r3 <- higher Fib. number
0e          01001011    4b    -- add r2,r3    ; r2 <- next Fib. number
0f          00011100    1c    -- ldi r3       ; r3 <- branch address X
10          00011010    1a    -- 1a           ; address X
11          10100011    a3    -- brn r3       ; if N, done - exit loop by branching to X
12          00000000    00    -- nop  ; branch delay slot
13          01010000    50    -- inc r0       ; increment memory pointer
14          01010100    54    -- inc r1       ; increment memory pointer
15          00111001    39    -- st r2,r1     ; store new Fib. number in RAM buffer
16          00011100    1c    -- ldi r3       ; r3 <- branch address L
17          00001100    0c    -- 0c           ; address L
18          10000011    83    -- jmp r3       ; jump to L (loop)
19          00000000    00    -- nop  ; branch delay slot

1a    X     00010000    10    -- ldi r0       ; r0 <- 80
1b          10000000    80    -- 80
1c          01110100    74    -- sub r1,r0    ; r1 has address of last Fib. number
1d          01010100    54    -- inc r1       ; get number of values stored in buffer
1e          00010000    10    -- ldi r0       ; r0 <- 8f (RAM address)
1f          10001111    8f    -- 8f
20          00110100    34    -- st r1,r0     ; M[8f] <- number of Fib. values in buffer

21    P     00010000    10    -- ldi r0       ; r0 <- 80 (beginning of RAM buffer)
22          10000000    80    -- 80
23          00010100    14    -- ldi r1       ; r1 <- 8f (address of number of values)
24          10001111    8f    -- 8f
25          00100101    25    -- ld r1,r1     ; r1 <- M[8f] (number of values in buffer)
26          01100101    65    -- neg r1,r1    ; r1 <- -r1 (two's complement)
27          00011000    18    -- ldi r2       ; r2 <- display port address
28          11111111    ff    -- ff

29    Q     00101100    2c    -- ld r3,r0     ; r3 <- Fib. number from RAM buffer
```

```
2a          00111110    3e    -- st r3,r2       ; display Fib. number on 7-segment displays
2b          00011100    1c    -- ldi r3         ; r3 <- branch address Z
2c          00110110    36    -- 36             ; address Z
2d          01010000    50    -- inc r0         ; increment RAM buffer pointer
2e          01010100    54    -- inc r1         ; increment loop counter
2f          00000000    00    -- nop  ; delay for Z, N status flags to be set
30          10010011    93    -- brz r3         ; if r1=0, branch to Z (exit loop)
31          00000000    00    -- nop  ; branch delay slot
32          00011100    1c    -- ldi r3         ; r3 <- branch address Q (top of loop)
33          00101001    29    -- 29             ; address Q
34          10000011    83    -- jmp r3         ; branch to Q
35          00000000    00    -- nop  ; branch delay slot

36    Z     00011100    1c    -- ldi r3         ; r3 <- branch address P
37          00100001    21    -- 21             ; address P
38          10000011    83    -- jmp r3         ; jump to P to re-display Fib. numbers
39          00000000    00    -- nop  ; branch delay slot
```