Problem M1.4: Microprogramming and Bus-Based Architectures [2 Hours]

In this problem, we explore microprogramming by writing microcode for the bus-based implementation of the MIPS machine described in Handout #4 (Bus-Based MIPS Implementation). Read the instruction fetch microcode in Table H4-3 which has been reproduced at the end of this problem (Worksheet M1-1) for the readers' convenience. Make sure that you understand how different types of data and control transfers are achieved by setting the appropriate control signals before attempting this problem.

In order to further simplify this problem, ignore the busy signal and assume that the memory is as fast as the register file.

The final solution should be elegant and efficient (e.g. number of new states needed, amount of new hardware added).

Problem M1.4.A Implementing Memory-to-Memory Add

For this problem, you are to implement a new memory-memory add operation. The new instruction has the following format.

ADDm r_d, r_s, r_t

ADDm performs the following operation.

$$M[r_d] \leftarrow M[r_s] + M[r_t]$$

Fill in Worksheet M1-1 with the microcode for ADDm. Use *don't cares* (*) for fields where it is safe to use don't cares. Study the hardware description well, and make sure all your microinstructions are legal.

Please comment your code clearly. If the pseudo-code for a line does not fit in the space provided, or if you have additional comments, you may write in the margins as long as you do it neatly. Your code should exhibit "clean" behavior and not modify any registers (except r_d) in the course of executing the instruction.

Finally, make sure that the instruction fetches the next instruction (by doing a microbranch to FETCH0 as discussed above).

Problem M1.4.B Implementing DBNEZ Instruction

DBNEZ stands for Decrease Branch Not Equal Zero. This instruction uses the same encoding as conditional branch instructions on MIPS.

6	5	5	16
opcode	**rs**		**Offset**

DBNEZ decrements register **rs** by 1, writes the result back to **rs** and branches to (**PC**+4)+**offset**, if result in **rs** is not equal to 0. Offset is sign extended to allow for backward branches. This instruction can be used for efficiently implementing loops.

Your task is to fill out Worksheet M1-2 for DBNEZ instruction. You should try to optimize your implementation for minimum number of cycles necessary and for maximum number of don't-care signals. You do not have to worry about the busy signal.

(Note that the microcode for the fetch stage has changed slightly from the one in Problem M1.4.A, to allow for a more efficient implementation of some instructions.)

Problem M1.4.C Implementing RETZ Instruction

In this question we ask you to implement a special return instruction, *return on zero* (`retz`), which uses the same encoding as a conditional branch instruction on MIPS.

`retz Rs, Rt`

6	5	5	16
`Retz`	**Rs**	**Rt**	**Unused**

`retz` instruction provides fast return from a subroutine call using **Rt** as the stack pointer. The instruction first tests the value of register **Rs**. If it is **not** zero, simply proceed to the next instruction at **PC+4**. If it is zero, the instruction does the following: (1) it reads the return address from memory at the address in register **Rt,** (2) increments **Rt** by 4 and (3) jumps to the return address.

Fill out Worksheet M1-3 for the `retz` instruction. You should try to optimize your implementation for minimum number of cycles necessary and for maximum number of don't-care signals. You do not have to worry about the busy signal. You may not need all the lines in the table for your solution.

You are allowed to introduce *at most* one new μBr target (Next State) for J (Jump) or Z (branch-if-Zero) other than FETCH0.

Problem M1.4.D **Implementing CALL Instruction**

In this question you will implement a new complex CALL instruction, which uses the same encoding as a conditional branch instruction on MIPS.

6	5	5	16
opcode	**ra**		**Offset**

CALL stores the return address, **PC**+4, to memory at the address in register **ra** (i.e., in **M[ra]**), decrements **ra** by 4, saves the new value back to **ra** and branches to (**PC**+4)+**offset**. This instruction provides fast subroutine calls, using register **ra** as the stack pointer.

Your task is to fill out Worksheet M1-4 for the CALL instruction. You should optimize your implementation to execute in the minimum number of cycles and to have the most signals set to don't care. You do not have to worry about the busy signal from memory. You may not need all the lines in the table for your solution.

Problem M1.4.E **Instruction Execution Times**

How many cycles does it take to execute the following instructions in the microcoded MIPS machine? Use the states and control points from MIPS-Controller-2 in Lecture 6 and assume Memory will not assert its busy signal.

Instruction	Cycles
`SUB  R3,R2,R1`	
`SUBI R2,R1,#4`	
`SW   R1,0(R2)`	
`BEQZ R1,label  # (R1 == 0)`	
`BNEZ R1,label  # (R1 != 0)`	
`J    label`	
`JR   R1`	
`JAL  label`	
`JALR R1`	

Which instruction takes the most cycles to execute? Which instruction takes the fewest cycles to execute?

Problem M1.4.F **Exponentiation**

Ben Bitdiddle needs to compute the power function for small numbers. Realizing there is no multiply instruction in the microcoded MIPS machine, he uses the following code to calculate the result when an unsigned number `m` is raised to the `n`th power, where `n` is another unsigned number.

```
if (m == 0) {
    result = 0;
}
else {
    result = 1;
    i = 0;

    while (i < n) {
        temp = result;
        j = 1;
        while (j < m) {
            result += temp;
            j++;
        }
        i++;
    }
}
```

The variables `i`, `j`, `m`, `n`, `temp` and `result` are unsigned 32-bit values.

Write the MIPS assembly that implements Ben's code. Use only the MIPS instructions that can be executed on the microcoded MIPS machine (ALU, ALUi, LW, SW, J, JAL, JR, JALR, BEQZ and BNEZ). The microcoded MIPS machine does not have branch delay slots. Use `R1` for `m`, `R2` for `n` and `R3` for `result`. At the end of your code only `R3` must have the correct value. The values of all other registers do not have to be preserved.

How many MIPS instructions are executed to calculate the power function? How many cycles does it take to calculate the power function? Again, use the states and control points from MIPS-Controller-2 and assume Memory will not assert its busy signal.

m, n	Instructions	Cycles
0, 1		
1, 0		
2, 2		
3, 4		
M, N		

Problem M1.4.G — Microcontroller Jump Logic

Now we will fill in a gap in the microcontroller implementation. In the lecture on microprogramming, we did not explain the implementation of the jump logic of the microcontroller. Your task in this problem is to implement that logic. Use AND gates, OR gates and inverters to implement the combinational logic that realizes the control equations for the jump logic of the MIPS microcontroller below. The control equations for the jump logic are

μPCSrc = *Case* μJumpTypes

next	**=>**	**μPC+1**
spin	**=>**	**μPC.busy + (μPC+1).~busy**
fetch	**=>**	**absolute**
dispatch	**=>**	**op-group**
feqz	**=>**	**absolute.zero + (μPC+1).~zero**
fnez	**=>**	**absolute.~zero + (μPC+1).zero**

The selection bits for each input of the μPCSrc mux, as well as the μJumpTypes encoding are given in the tables below. Your task is to create combinational logic that translates between them, according to the control equations. Assume that the busy and zero signals follow positive logic (so they are true if the wire is carrying a 1 and false if the wire is carrying a 0). Your design will be judged for its correctness, clarity and organization. These factors are more important than the efficiency of your design.

μJumpTypes	Encoding
next	000
spin	001
feqz	110
fnez	111
fetch	010
dispatch	100

Table M1.4-1: μJumpTypes Encoding

μPCSrc	Selection bits
μPC+1	00
μPC	01
absolute	10
op-group	11

Table M1.4-2: μPCSrc Selection bits

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	0	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	0	0	1	*	0	N	*
	PC <- A+4	0	PC	1	1	0	*	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDM0:																

Worksheet M1-1

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
DBNEZ:																

Worksheet M1-2

State	PseudoCode	Ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Im m	μBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
`retz0`																

Worksheet M1-3

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Mem	Ex Sel	en Imm	µBr	Next State
FETCH0:	MA <- PC; A <- PC	*	PC	0	1	1	*	*	0	1	*	0	*	0	N	*
	IR <- Mem	1	*	*	0	0	*	*	0	*	0	1	*	0	N	*
	PC <- A+4; B <- A+4	0	PC	1	1	*	1	INC_A_4	1	*	*	0	*	0	D	*
. . .																
NOP0:	microbranch back to FETCH0	*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
CALL:																

Worksheet M1-4