

Problem M1.4: Microprogramming and Bus-Based Architectures

Problem M1.4.A

Memory-to-Memory Add

Worksheet M1-1 shows one way to implement ADDm in microcode.

Note that to maintain “clean” behavior of your microcode, no registers in the register file should change their value during execution (unless they are written to). This does not refer to the registers in the datapath (IR, A, B, MA). Thus, using asterisks for the load signals (ldIR, ldA, ldB, and ldMA) is acceptable as long as the correctness of your microcode is not affected.

Problem M1.4.B

Implementing DBNEZ Instruction

The question asked to jump to PC+4+offset. This ignores that the immediate value needs to be shifted left by 2 before it can be added to PC+4, to make sure we don’t run into alignment problems. We did this because the data path given doesn’t really have facilities for shifting.

Worksheet M1-2 shows one way to implement DBNEZ in microcode.

Problem M1.4.C

Implementing RETZ Instruction

Worksheet M1-3 shows one way to implement RETZ in microcode.

Problem M1.4.D

Implementing CALL Instruction

Worksheet M1-3 shows one way to implement CALL in microcode.

Problem M1.4.E

Instruction Execution Times

Instruction	Cycles
SUB R3,R2,R1	3 + 3 = 6
SUBI R2,R1,#4	3 + 3 = 6
SW R1,0(R2)	3 + 5 = 8
BNEZ R1,label # (R1 == 0)	3 + 2 = 5
BNEZ R1,label # (R1 != 0)	3 + 5 = 8
BEQZ R1,label # (R1 == 0)	3 + 5 = 8
BEQZ R1,label # (R1 != 0)	3 + 2 = 5
J label	3 + 3 = 6
JR R1	3 + 2 = 5
JAL label	3 + 4 = 7
JALR R1	3 + 4 = 7

As discussed in Lecture 6, instruction execution includes the number of cycles needed to fetch the instruction. The lecture notes used 4 cycles for the fetch phase, while Worksheet 1 shows that this phase can actually be implemented in 3 cycles—either answer is fine. The above table uses 3 cycles for the fetch phase. Overall, SW, BNEZ (for a taken branch), and BEQZ (for a taken branch) take the most cycles to execute (8), while BNEZ (for a not-taken branch), BEQZ (for a not-taken branch) and JR take the fewest cycles (5).

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	1	*	0	N	*
	PC <- A+4; dispatch	0	PC	1	1	*	*	INC_A_4	1	*	*	0	*	0	D	*
...																
NOP0:	microbranch Back to FETCH0	0	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
ADDm0:	MA <- R[rs]	0	rs	0	1	*	*	*	0	1	*	0	*	0	N	*
	A <- Mem	0	*	*	0	1	*	*	0	*	0	1	*	0	N	*
	MA <- R[rt]	0	rt	0	1	0	*	*	0	1	*	0	*	0	N	*
	B <- Mem	0	*	*	0	0	1	*	0	*	0	1	*	0	N	*
	MA <- R[rd]	*	rd	0	1	0	0	*	0	1	*	0	*	0	N	*
	Mem <- A+B; fetch	*	*	*	0	*	*	ADD	1	*	1	1	*	0	J	FETCH0

Worksheet M1-1: Implementation of the ADDm instruction

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:	A <- rs	0	rs	0	1	1	0	*	0	*	*	0	*	0	N	*
	rs <- A – 1 μB to FETCH0 if zero	0	rs	1	1	*	0	DEC_A_1	1	*	*	0	*	0	Z	FETCH0
	A <- sExt16(IR)	*	*	*	0	1	0	*	0	*	*	0	sExt16	1	N	*
	PC <- A+B jump to FETCH0	*	PC	1	1	*	*	ADD	1	*	*	0	*	0	J	FETCH0

Worksheet M1-2: Implementation of the DBNEZ Instruction

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	A <- Reg[Rs]	0	Rs	0	1	1	*	*	0	*	*	0	*	0	N	*
retz1	A <- Reg[Rt] MA <- Reg[Rt] uBr to retz3 if zero	0	Rt	0	1	1	*	COPY_A	0	1	*	0	*	0	Z	retz3
retz2		*	*	*	0	*	*	*	0	*	*	0	*	0	J	FETCH0
retz3	PC <- MEM	0	PC	1	1	0	*	*	0	*	0	1	*	0	N	*
retz4	Reg[Rt] < A+4	*	Rt	1	1	*	*	INC_A_4	1	*	*	0	*	0	J	FETCH0

Worksheet M1-3: Implementation of the RETZ Instruction

State	PseudoCode	ld IR	Reg Sel	Reg W	en Reg	ld A	ld B	ALUOp	en ALU	Ld MA	Mem W	en Me m	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:	MA <- R[ra]; A <- R[ra]	0	ra	0	1	1	0	*	0	1	*	0	*	0	N	*
	Mem <- B	0	*	*	0	0	0	COPY_B	1	*	1	1	*	0	N	*
	R[ra] <- A - 4	0	ra	1	1	*	0	DEC_A_4	1	*	*	0	*	0	N	*
	A <- sExt16(IR)	*	*	*	0	1	0	*	0	*	*	0	sExt16	1	N	*
	PC <- A+B; jump to FETCH0	*	PC	1	1	*	*	ADD	1	*	*	0	*	0	J	FETCH0

Worksheet M1-4: Implementation of the CALL Instruction

Problem M1.4.F**Exponentiation**

In the given code, ‘m’ and ‘n’ are always nonnegative integers. Therefore, we don’t have to worry about the cases where ‘i’ is larger than ‘n’ or ‘j’ is larger than ‘m’. Also, for this problem, 0 raised to any power is just 0, while any nonzero value raised to the 0th power is 1. Note that the pseudo code that is given returns a value of 0 when 0 is raised to the 0th power. However, the actual `pow()` function in the standard C library returns a value of 1 for this case. We present the solution that implements the pseudo code given in the problem rather than C’s `pow()` function.

```
#
# R5: temp, R6: j
#

        ADD    R3, R0, R0        ; put 0 in result
        BEQZ   R1, _END_I        ; if m is 0, end
        ADDI   R3, R0, #1        ; put 1 in result
        BEQZ   R2, _END_I        ; if n is 0, the loop is over; we set
                                ; i equal to n and count down to 0—since
                                ; R2 does not have to be preserved, we
                                ; use it for i
        SUBI   R5, R1, #1        ; temp = m - 1
        BEQZ   R5, _END_I        ; if m is 1, the result will be 1,
                                ; so end the program

_START_I:
        ADD    R5, R0, R3        ; temp = result
        SUBI   R6, R1, #1        ; j = m - 1 (the number of times to
                                ; execute the second loop)

_START_J:
        ADD    R3, R3, R5        ; result += temp
        SUBI   R6, R6, #1        ; j--
        BNEZ   R6, _START_J      ; Re-execute loop until j reaches 0

_END_J:
        SUBI   R2, R2, #1        ; i--
        BNEZ   R2, _START_I      ; Re-execute loop until i reaches 0

_END_I:
```

To compute the number of instructions and cycles to execute this code, let us consider subsets of the code.

Code	# of instructions	# of cycles
ADD R3, R0, R0 BEQZ R1, _END_I	2	$6 \times 1 + 8 \times 1 = 14$ (m = 0) $6 \times 1 + 5 \times 1 = 11$ (m > 0)
ADDI R3, R0, #1 BEQZ R2, _END_I	2 (if m > 0)	$6 \times 1 + 8 \times 1 = 14$ (n = 0) $6 \times 1 + 5 \times 1 = 11$ (n > 0)
SUBI R5, R1, #1 BEQZ R5, _END_I	2 (if m > 0 and n > 0)	$6 \times 1 + 8 \times 1 = 14$ (m = 1) $6 \times 1 + 5 \times 1 = 11$ (m > 1)
_START_I: ADD R5, R0, R3 SUBI R6, R1, #1	2n (if m > 1 and n > 0)	$(6 \times 2) \times n = 12n$
_START_J: ADD R3, R3, R5 SUBI R6, R6, #1 BNEZ R6, _START_J	$3n(m-1)$ (if m > 1 and n > 0)	$(6 \times 2 + 5 \times 1) \times n + (6 \times 2 + 8 \times 1) \times (m-2) \times n = 17n + 20n(m-2)$
_END_J: SUBI R2, R2, #1 BNEZ R2, _START_I	2n (if m > 1 and n > 0)	$(6 + 8) \times n - 3 = 14n - 3$

From the above table, we can complete the table given in the problem.

m,n	Instructions	Cycles
0, 1	2	14
1, 0	4	25
2, 2	20	116
3, 4	46	282
M, N (M = 0)	2	14
M, N (M > 0, N = 0)	4	25
M, N (M = 1, N > 0)	6	36
M, N (M > 1, N > 0)	3N(M-1)+4N+6	20N(M-2)+43N+30

Problem M1.4.G

Microcontroller Jump Logic

One way to start designing the microcontroller jump logic is to write out a table of the input signals and the output bits. For clarity, the bits that encode the μ JumpTypes are labeled A, B and C, from left to right. The output bits are labeled H and L, also from left to right. So the table we need to implement is the following (where asterisks are for the input bits that we don't care about).

Input bits					Output bits	
A	B	C	Zero	Busy	H	L
0	0	0	*	*	0	0
0	0	1	*	0	0	0
0	0	1	*	1	0	1
0	1	0	*	*	1	0
1	0	0	*	*	1	1
1	1	0	0	*	0	0
1	1	0	1	*	1	0
1	1	1	0	*	1	0
1	1	1	1	*	0	0

Writing out boolean equations for the H and L output bits (by directly recognizing only the lines which have logical ones as output) we find

$$H = \overline{A}\overline{B}\overline{C} + \overline{A}B\overline{C} + \overline{A}BC \cdot \overline{zero} + ABC \cdot \overline{zero}$$

$$L = \overline{A}BC \cdot busy + \overline{A}B\overline{C}$$

Also, we do not care about the output when the μ Jump type is 011 or 101, since those are invalid encodings. Thus we can simplify the equations to

$$H = \overline{A}\overline{B} + \overline{A}B + \overline{A}\overline{C} \cdot \overline{zero} + AC \cdot \overline{zero}$$

$$L = \overline{B}C \cdot busy + \overline{A}\overline{B}$$

Drawing this out as gates we get

