CMU 18-447 Introduction to Computer Architecture, Spring 2015

Midterm Exam 1

Date: Fri., 3/20

Instructor: Onur Mutlu
TAs: Rachata Ausavarungnirun, Kevin Chang, Albert Cho, Jeremie Kim, Clement Loh

Name:

Problem 1 (80 Points):	
Problem 2 (40 Points):	
Problem 3 (40 Points):	
Problem 4 (60 Points):	
Problem 5 (30 Points):	
Problem 6 (30 Points):	
Problem 7 (60 Points):	
Bonus (60 Points):	
Total (340 + 60 Points):	

Instructions:

1. This is a closed book exam. You are allowed to have one letter-sized cheat sheet.
2. No electronic devices may be used.
3. This exam lasts 1 hour and 50 minutes.
4. Clearly indicate your final answer for each problem.
5. Please show your work when needed.
6. Please write your initials at the top of every page.
7. Please make sure that your answers to all questions (and all supporting work that is required) are contained in the space required.

Tips:

- **Be cognizant of time.** Do not spend too much time on one question.
- **Be concise.** You will be penalized for verbosity and unnecessarily long answers.
- **Show work when needed.** You will receive partial credit at the instructors' discretion.
- **Write legibly.** Show your final answer.

1. Potpourri [80 points]

(a) For each of the following, circle if the concept is a part of the ISA versus the microarchitecture (circle only one):

Number of threads in fine-grained multithreading
Circle one: ISA Microarchitecture

Number of DRAM banks
Circle one: ISA $Microarchitecture$

Vectored interrupts
Circle one: ISA Microarchitecture

Number of entries in reservation stations
Circle one: ISA $Microarchitecture$

Number of entries in the reorder buffer
Circle one: ISA $Microarchitecture$

Number of entries in the architectural register file
Circle one: ISA Microarchitecture

Number of entries in the physical register file
Circle one: ISA $Microarchitecture$

Number of sets in the L3 cache
Circle one: ISA $Microarchitecture$

The page table base register of the executing process
Circle one: ISA Microarchitecture

(b) Why does ISA change more slowly than microarchitecture?

> ISA is exposed to the programmer. Adopting a new or modified ISA requires changes to software and compilers, whereas adopting a new microarchitecture requires no such changes.

(c) A program is written in C. We execute this program on two different computers:

Computer A: has a processor that implements the x86 ISA and has 3 GHz clock frequency
Computer B: has a processor that implements the x86 ISA and has 3 GHz clock frequency

When we execute this program and measure its cycles per instruction (CPI) in x86 instructions, we find the following result:

On Computer A: CPI is equal to 10
On Computer B: CPI is equal to 8

What can you say about on which computer (A or B) this program runs faster?

We don't know.

Explain and show all your work below:

Because we don't know how many instructions are actually executed for the program on either machine, we cannot conclude which computer runs faster. Although B has lower CPI, but it might be executing 2 times more instructions than A due to a less optimized compiler.

(d) You are designing an ISA that uses delayed branch instructions. You are trying to decide how many instructions to place into the branch delay slot. How many branch delay slots would you need for the following different implementations? Explain your reasoning briefly.

An in-order processor where conditional branches resolve during the 4th stage:

3

An out-of-order processor with 64 unified reservation station entries where conditional branches resolve during the 2nd cycle of branch execution. The processor has 15 pipeline stages until the start of the execution stages.

We don't know.

(e) What three key pieces of information does the compiler not know when performing instruction scheduling?

Memory addresses

Cache hit/miss status

Branch direction

(f) In class, we discussed the concept of traces and trace scheduling.

What is a trace?

A frequently executed sequence of basic blocks.

Now suppose we make each trace atomic, as we also discussed in class. What is the benefit of making each trace atomic? Explain.

Enables more compiler optimizations. The compiler can freely reorder instructions within the atomic trace subject only to true dependencies, without requiring any fix-up code.

What is the disadvantage?

Wasted work when the atomic trace that is executed is not for the control flow path that is supposed to be executed.

(g) Assume we have an ISA with virtual memory. It is byte-addressable and its address space is 64 bits. The physical page size is 8KB. The size of the physical memory we use in a computer that implements the ISA is 1 TB (2^{40} bytes).

Assume the demand paging system we would like to design for this uses the perfect LRU algorithm to decide what to evict on a page fault. What is the minimum number of bits that the operating system needs to keep to implement this perfect LRU algorithm? Show your work.

$\lceil \log_2 2^{27}! \rceil$

2. Register Renaming [40 points]

In this problem, we will give you the state of the Register Alias Table (RAT), Reservation Stations (RS), and Physical Register File (PRF) for a Tomasulo-like out-of-order execution engine.

The out-of-order machine in this problem has the following characteristics:

- The processor is fully pipelined with four stages: Fetch, decode, execute, and writeback.
- For all instructions, fetch takes 1 cycle, decode takes 1 cycle, and writeback takes 1 cycle.
- The processor implements `ADD` and `MUL` instructions only. Both the adder and multiplier are fully pipelined. `ADD` instructions take 3 cycles and `MUL` instructions take 4 cycles in the execute stage. Note that the adder and multiplier have separate common data buses (CDBs), which allow both the adder and multiplier to broadcast results in the same cycle.
- An instruction always allocates the first reservation station that is available (in top-to-bottom order) at the required functional unit.

Suppose the pipeline is initially empty and the machine fetches exactly 5 instructions. The diagram below shows the snapshot of the machine at a particular point in time.

Register Alias Table

ID	V	Tag
R0	1	P0
R1	1	P8
R2	0	P15
R3	1	P3
R4	0	P10
R5	1	P5
R6	1	P12
R7	1	P7

Physical Register File

ID	V	Data	ID	V	Data
P0	1	1	P8	1	87
P1	0	10	P9	1	90
P2	0	2	P10	1	11
P3	1	30	P11	0	110
P4	0	3	P12	1	37
P5	1	50	P13	0	130
P6	0	5	P14	1	17
P7	1	70	P15	1	159

`ADD` **Reservation Station**

ID	V	Tag	V	Tag	Dest. Tag
A	0	–	0	–	–
B	1	P15	1	P5	P14

`ADD` **CDB**

`MUL` **Reservation Station**

ID	V	Tag	V	Tag	Dest. Tag
X	1	P8	1	P12	P15
Y	1	P8	1	P15	P9
Z	0	P14	0	P14	P10

`MUL` **CDB**

(a) Your first task is to use only the supplied information to draw the data flow graph for the five instructions which have been fetched. Label nodes with the operation (+ or $\star$) being performed and edges with the architectural register alias numbers (e.g., R0).

(b) Now, use the data flow graph to fill in the table below with the five instructions being executed on the processor in program order. The source registers for the first instruction can be specified in either order. Give instructions in the following format: "opcode, destination, source1, source2."

OP	Dest	Src 1	Src 2
ADD	R1	R5	R6
MUL	R2	R1	R6
MUL	R4	R2	R1
ADD	R4	R5	R2
MUL	R4	R4	R4

(c) Now show the full pipeline timing diagram below for the sequence of five instructions that you determined above, from the fetch of the first instruction to the writeback of the last instruction. Assume that the machine stops fetching instructions after the fifth instruction.

As we saw in class, use F for fetch, D for decode, En to signify the nth cycle of execution for an instruction, and W to signify writeback. You may or may not need all columns shown. Finally, identify the cycle after which the snapshot of the microarchitecture was taken. Shade the corresponding cycle in the last row of the table.

Cycle:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Instruction 1	F	D	E1	E2	E3	W													
Instruction 2		F	D			E1	E2	E3	E4	W									
Instruction 3			F	D						E1	E2	E3	E4	W					
Instruction 4				F	D					E1	E2	E3	W						
Instruction 5					F	D							E1	E2	E3	E4	W		
Snapshot cycle									X										

3. Branch Prediction [40 points]

Assume the following piece of code that iterates through two large arrays, j and k, each populated with completely (i.e., truly) random positive integers. The code has two branches (labeled `B1` and `B2`). When we say that a branch is *taken*, we mean that the code *inside* the curly brackets is executed. Assume the code is run to completion without any errors (there are no exceptions). For the following questions, assume that this is the only block of code that will ever be run, and the loop-condition branch (`B1`) is resolved first in the iteration before the if-condition branch (`B2`). `N` and `X` are unspecified non-zero integers.

```
for (int i = 0; i < N; i++) {      /* B1 */
                                   /* TAKEN PATH for B1 */
    if (i % X == 0) {              /* B2 */
        j[i] = k[i] - i;           /* TAKEN PATH for B2 */
    }
}
```

You are running the above code on a machine with a two-bit global history register (GHR) shared by all branches, which starts at *Strongly Not Taken* (2'b00). Each pattern history table entry (PHTE) contains a 2-bit saturating counter, which is initialized to *Strongly Taken* (2'b11)
The saturating counter values are as follows:
2'b00 - Strongly Not Taken
2'b01 - Weakly Not Taken
2'b10 - Weakly Taken
2'b11 - Strongly Taken

(a) Assuming that N is larger than 10 (ten), after running the loop for 10 iterations, you observe that the branch predictor mispredicts 0% of the time. What is the value of X? Explain your reasoning.

$X = 1$
In this case, the branch predictor is always right. Because all the counters in the PHTE start at Strongly Taken, it will have to mispredict at least once if the branch B2 is not taken at least once. This means B2 has to always result in a taken outcome.

(b) What is the prediction accuracy of the branch predictor if $N = 20$ and $X = 2$? Explain your answer. You can leave your answer as a fraction.

Mispredict 11 times out of 41 branches (The first ten times when there is a not taken occurs in B2, GHR entries for 11 got updated to Weakly taken, and one more mispredict branch when B1 terminates).
Accuracy = 30/41 = 73.2%

(c) Assuming that N is larger than 10 (ten), after running this code until it exits the loop, you observe that the state of the branch predictor is as follows:

GHR: 2'b00

PHTE:
NN:2'b11
NT:2'b00
TN:2'b10
TT:2'b11

What can you tell about the value of X and N? Explain your reasoning.

$X >= N$ or $N \mod X! = 1$
If $X > 1$, the program will always execute with the following pattern: TTTNT-NTNT...TNTTTNTNTNTN... where the TN pattern will happen based on the value of X, and at the exit of the loop the pattern will end with N. In this question, the only way the state of NT to have a value greater than zero is when the last iteration B2 is not taken.

4. GPUs and SIMD [60 points]

We define the *SIMD utilization* of a program running on a GPU as the fraction of SIMD lanes that are kept busy with *active threads*.

The following code segment is run on a GPU. Each thread executes **a single iteration** of the shown loop. Assume that the data values of the arrays, A and B, are already in the vector registers so there are no loads and stores in this program. Hint: Notice that there are 2 instructions in each thread. A warp in this GPU consists of 32 threads, and there are 32 SIMD lanes in the GPU. Assume that each instruction takes the same amount of time to execute.

```
for (i = 0; i < N; i++) {
    if (A[i] % 3 == 0) {     // Instruction 1
       A[i] = A[i] * B[i];   // Instruction 2
    }
}
```

(a) What's the minimum number of bits required to encode the warp ID in order to execute this program? Please leave the answer in terms of N.

$\lceil log_2(\frac{N}{32}) \rceil$

(b) Assume integer array A has a repetitive pattern which consists of 24 ones followed by 8 zeros, and integer array B has a different repetitive pattern which consists of 48 zeros followed by 64 ones. What is the SIMD utilization of this program?

((24+8*2)/(32*2))*100% = 40/64*100 = 62.5%

(c) Is it possible for this program to yield a SIMD utilization of 100% (circle one)?

YES NO

If YES, what should be true about arrays A for the SIMD utilization to be 100%?

Yes. If, for every 32 elements of A, all of them are divisible by 3, or if all are not divisible by 3.

If NO, explain why not.

What should be true about array B?

B can be any array of integers.

(d) Is it possible for this program to yield a SIMD utilization of 56.25% (circle one)? *Hint: 56.25% = 36/64.*

YES NO

If YES, what should be true about arrays A for the SIMD utilization to be 56.25%?

Yes, if 4 out of every 32 elements of A are divisible by 3.

What should be true about arrays B?

B can be any array of integers.

If NO, explain why not.

(e) Is it possible for this program to yield a SIMD utilization of 50% (circle one)?

YES NO

If YES, what should be true about arrays A for the SIMD utilization to be 50%?

What should be true about arrays B?

If NO, explain why not.

No. The minimum is where 1/32 elements in array A are even. This yields a 51.5625% usage.

In lecture, we learned a technique called dynamic warp formation, which tries to improve SIMD utilization, by merging threads executing the same instruction together. The key idea of dynamic warp formation is to move an "active" thread from one warp to another warp such that SIMD utilization is improved and all threads can access their registers.

Consider the following three warps X, Y, and Z that are executing the same code segment specified in the beginning of this question. Assume that the vectors we provide below specify the "active mask", i.e., whether or not the instruction should be executed by each thread in the warp: 1 means the instruction should be executed, 0 means it should not be executed. Assume each warp is at the same Program Counter.

```
Warp X = {10000000000000000000000000000010}
Warp Y = {10000000000000000000000000000001}
Warp Z = {01000000000000000000000000000000}
```

(f) Suppose that you perform dynamic warp formation on these three warps. What are the resulting active masks for each of the newly formed warps X', Y', and Z'? Explain your reasoning as necessary.

There are several answers for this question but the key is that the taken branch in Z can be combined with either X or Y. However, the taken branch in the first thread of X and Y cannot be merged because they are on the same GPU lane.
X = 10000000000000000000000000000010
Y = 11000000000000000000000000000001
Z = 00000000000000000000000000000000

(g) Given the original specification for arrays A and B (integer array A has a repetitive pattern which consists of 24 ones followed by 8 zeros, and integer array B has a different repetitive pattern which consists of 48 zeros followed by 64 ones). Is it possible for this program to yield a better SIMD utilization than the baseline when dynamic warp formation is used?

No. Branch divergence happens on the same lane throughout the program.

5. Caches [30 points]

A byte-addressable system with 16-bit addresses ships with a three-way set associative, write-back cache. The cache implements a true LRU replacement policy using the minimum number of replacement policy bits necessary to implement it. The tag store requires a total of 264 bits of storage. What is the block size of the cache? (Hint: $264 = 2^8 + 2^3$)

Answer:

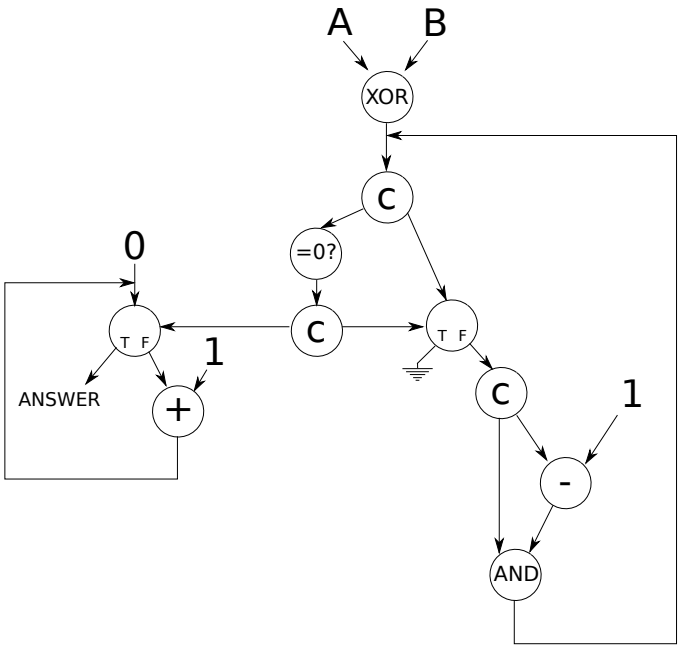
2^5 bytes

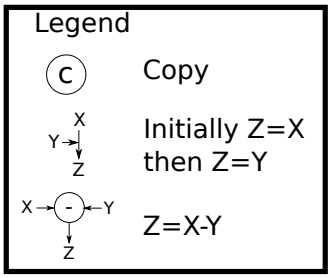
Show all your work.

Assume t tag bits, n index bits and b block bits.
$t + n + b = 16$
LRU = $\lceil \log_2 3! \rceil = 3$ bits per set with 3-way associativity
Valid Bit = 1 bit per block
Dirty Bit = 1 bit per block
Tag bits = t bits per block
Number of sets = 2^n
Number of blocks = $3 * 2^n$ (3-way associative)
Tag store size = $3 * 2^n + 3 * 2^n * (2 + t) = 2^n * (9 + 3t)$
We get $2^n * (9 + 3t) = 2^8 + 2^3$
$2^n * (9 + 3t) = 2^3 * (2^5 + 1)$
$2^n * (9 + 3t) = 2^3 * 33$
So, $n = 3$ and $t = 8$. As a result, $b = 5$
Therefore, the block size is 2^5.

6. Dataflow [30 points]

Here is a dataflow graph representing a dataflow program:





Note that the inputs, A and B, are non-negative integers.

What does the dataflow program do? Specify clearly in less than 15 words.

Calculates the Hamming distance of A and B.

7. VLIW and Instruction Scheduling [60 points]

Explain the motivation for VLIW in one sentence.

> Enable multiple instruction issue with simple hardware. Independent instructions can be statically scheduled into a single VLIW instruction that can fed into multiple functional units concurrently.

You are the human compiler for a VLIW machine whose specifications are as follows:

- There are 3 **fully** pipelined functional units (ALU, MU and FPU).
- Integer Arithmetic Logic Unit (ALU) has a 1-cycle latency.
- Memory Unit (MU) has a 2-cycle latency.
- Floating Point Unit (FPU) has a 3-cycle latency, and can perform either FADD or FMUL (floating point add / floating point multiply) on **floating point registers**.
- This machine has **only** 4 integer registers (r1 .. r4) and 4 floating point registers (f1 .. f4)
- The machine does not implement hardware interlocking or data forwarding.

(a) For the given assembly code on the next page, fill **Table 1** (on the next page) with the appropriate VLIW instructions for only one iteration of the loop (The C code is also provided for your reference). Provide the VLIW instructions that lead to the **best** performance. Use the minimum number of VLIW instructions. Table 1 should **only** contain instructions provided in the assembly example. For all the instruction tables, show the NOP instructions you may need to insert. Note that `BNE` is executed in the **ALU**.

The base addresses for A, B, C are stored in r1, r2, r3 respectively. The address of the last element in the array C[N-1] is stored in r4, where N is an integer multiplier of 10! (read: 10 factorial).

Three extra tables are available for you to work with in the scratchpad, for the entirety of this question.

C code

```
float A[N];
float C[N];
int B[N];
... //code to initialize A and B
for (int i=0; i<N; i++)
        C[i] = A[i] * A[i] +B[i];
```

Assembly Code

```
loop:   LD      f1, 0 (r1)
        LD      f2, 0 (r2)
        FMUL    f1, f1, f1
        FADD    f1, f1, f2
        ADDI    r3, r3, 4
        ST      f1, -4 (r3)
        ADDI    r1, r1, 4
        ADDI    r2, r2, 4
        BNE     r3, r4, loop
```

VLIW Instruction	**ALU**	**MU**	**FPU**
1	ADDI r1, r1, 4	LD f1, 0(r1)	NOP
2	ADDI r2, r2, 4	LD f2, 0(r2)	NOP
3	NOP	NOP	FMUL f1, f1, f1
4	NOP	NOP	NOP
5	NOP	NOP	NOP
6	NOP	NOP	FADD f1, f1, f2
4	NOP	NOP	NOP
8	ADDI r3, r3, 4	NOP	NOP
9	BNE r3, r4, loop	ST f1, -4(r3)	NOP
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Table 1

What is the performance in Ops/VLIW instruction (Operations/VLIW instruction) for this design? An operation here refers to an instruction (in the Assembly Code), excluding NOPs.

1

(b) Assume now we decide to unroll the loop once. Fill **Table 2** with the new VLIW instructions. You should optimize for latency first, then instruction count. **You can choose to use different offsets, immediates and registers, but you may not use any new instructions.**

VLIW Instruction	**ALU**	**MU**	**FPU**
1	NOP	LD f1, 0(r1)	NOP
2	ADDI r1, r1, 8	LD f3, 4(r1)	NOP
3	NOP	LD f2, 0(r2)	FMUL f1, f1, f1
4	ADDI r2, r2, 8	LD f4, 4(r2)	FMUL f3, f3, f3
5	NOP	NOP	NOP
6	NOP	NOP	FADD f1, f1, f2
7	NOP	NOP	FADD f3, f3, f4
8	NOP	NOP	NOP
9	ADDI r3, r3, 8	ST f1, 0(r3)	NOP
10	BNE r3, r4, loop	ST f3, -4(r3)	NOP
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Table 2

What is the performance in Ops/VLIW instruction for this design?

14/10

(c) Assume now we have **unlimited registers** and the loop is fully optimized (unrolled to the **best** performance possible). What is the performance in Ops/cycle for this design? Show your work and explain **clearly** how you arrived at your answer. You are not required to draw any tables, but you may choose to do so to aid your explanation. You will receive **zero** credit for a correct answer without any explanation. (Hint: trace the dependent instructions)

29/15. Notice that we can add 3 MU ops (2 LDs and 1 ST) and 2 FPU ops per unroll, while the ALU ops remain constant at 4. If you trace the table carefully, you will observe that the MU instruction stream will have 1 op/cycle by the time we unroll the loop five times. At this point, we have $4 + 15 + 10 = 29$ instructions over 15 cycles. Any further unrolling will result in a smaller ops/cycle since the MU instruction stream is already saturated.

What is the performance bottleneck for this code and why? Explain.

Memory Unit. We add 3 MU ops but only 2 FPU ops per unroll, eventually causing the MU instruction stream to be saturated before the FPU, despite the FPU having a longer compute latency. (We are more concerned with throughput than latency)

8. [Bonus] Mystery Instruction [60 points]

A pesky engineer implemented a mystery instruction on the LC-3b. It is your job to determine what the instruction does. The mystery instruction is encoded as:

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
1010				DestR			SR1			0	0	0	SR2		

The instruction is only defined if the value of SR2 is greater than the value of SR1.

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the four pages at the end of this question). We also provide the original LC-3b state diagram, in case you need it. (As a reminder, the selection logic for SR2MUX is determined internally based on the instruction.)

The additional control signals are

LD_TEMP1/1: NO, YES

LD_TEMP2/1: NO, YES

GateTEMP3/1: NO, LOAD

Reg_IN_MUX/1: BUS, Mystery2 – (Assume BUS is asserted if this signal is not specified)

Mystery_MUX/2: SR2MUX, PASS_1 (outputs value 1), PASS_0 (outputs value 0)

Additional Signals for ALUK: PASS_B (outputs the value from input B), SUB (A-B)

Also note that both of DRMUX and SR1MUX can now choose DR, SR1, and SR2

COND/4:
$COND_{0000}$;Unconditional
$COND_{0001}$;Memory Ready
$COND_{0010}$;Branch
$COND_{0011}$;Addressing mode
$COND_{0100}$;Mystery 1
$COND_{1000}$;Mystery 2

The microcode for the instruction is given in the table below.

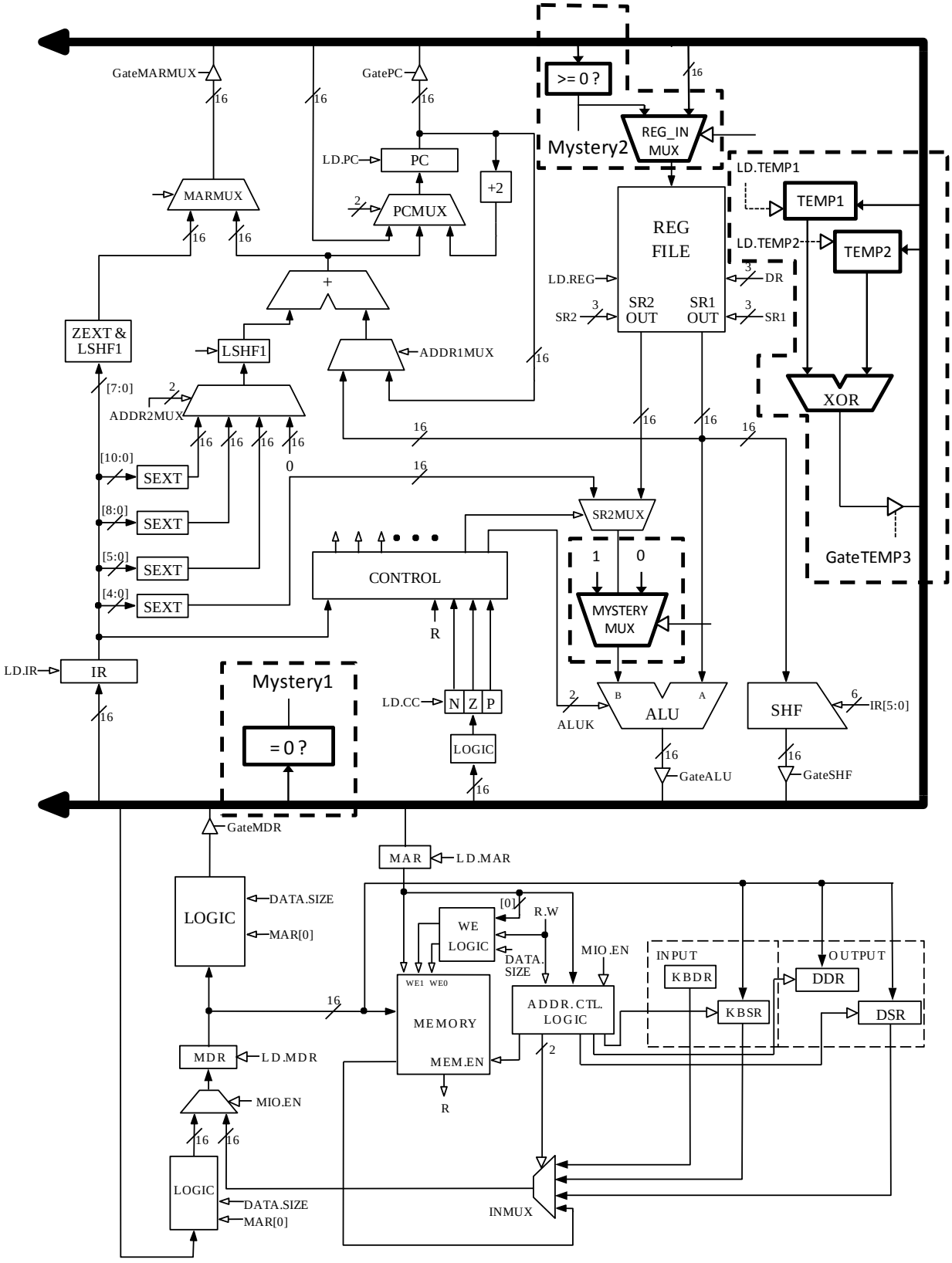
State	Cond	J	Asserted Signals
001010 (10)	$COND_{0000}$	001011	LD.REG, DRMUX = DR(IR [11:9]), GateALU, ALUK = PASS_B, MYSTERY_MUX = PASS_0
001011 (11)	$COND_{0000}$	110001	LD.MAR, SR1MUX = SR1(IR[8:6]), ADDR1MUX = SR1OUT, ADDR2MUX = 0, MARMUX = ADDER, GateMARMUX
110001 (49)	$COND_{0001}$	110001	LD.MDR, MIO.EN, DATA.SIZE=BYTE, R.W = R
110011 (51)	$COND_{0000}$	100100	GateMDR, LD.TEMP1, DATA.SIZE=BYTE
100100 (36)	$COND_{0000}$	100101	LD.MAR, SR1MUX = SR2(IR[2:0]), ADDR1MUX = SR1OUT, ADDR2MUX = 0, MARMUX = ADDER, GateMARMUX
100101 (37)	$COND_{0001}$	100101	LD.MDR, MIO.EN, DATA.SIZE=BYTE, R.W = R
100111 (39)	$COND_{0000}$	101000	GateMDR, LD.TEMP2, DATA.SIZE=BYTE
101000 (40)	$COND_{0100}$	010010	GateTEMP3
110010 (50)	$COND_{0000}$	101001	LD.REG, DRMUX = SR1(IR[8:6]), GateALU, ALUK = ADD, SR1MUX = SR1(IR[8:6]), MYSTERY_MUX = PASS_1
101001 (41)	$COND_{0000}$	101010	LD.REG, DRMUX = SR2(IR[2:0]), GateALU, ALUK = SUB, SR1MUX = SR2 (IR[2:0]), MYSTERY_MUX = PASS_1
101010 (42)	$COND_{1000}$	001011	LD.REG, DRMUX = DR (IR[11:9]), Reg_IN_MUX = MYSTERY2, GateALU, ALUK = SUB, SR1MUX = SR1(IR[8:6]), MYSTERY_MUX = SR2MUX

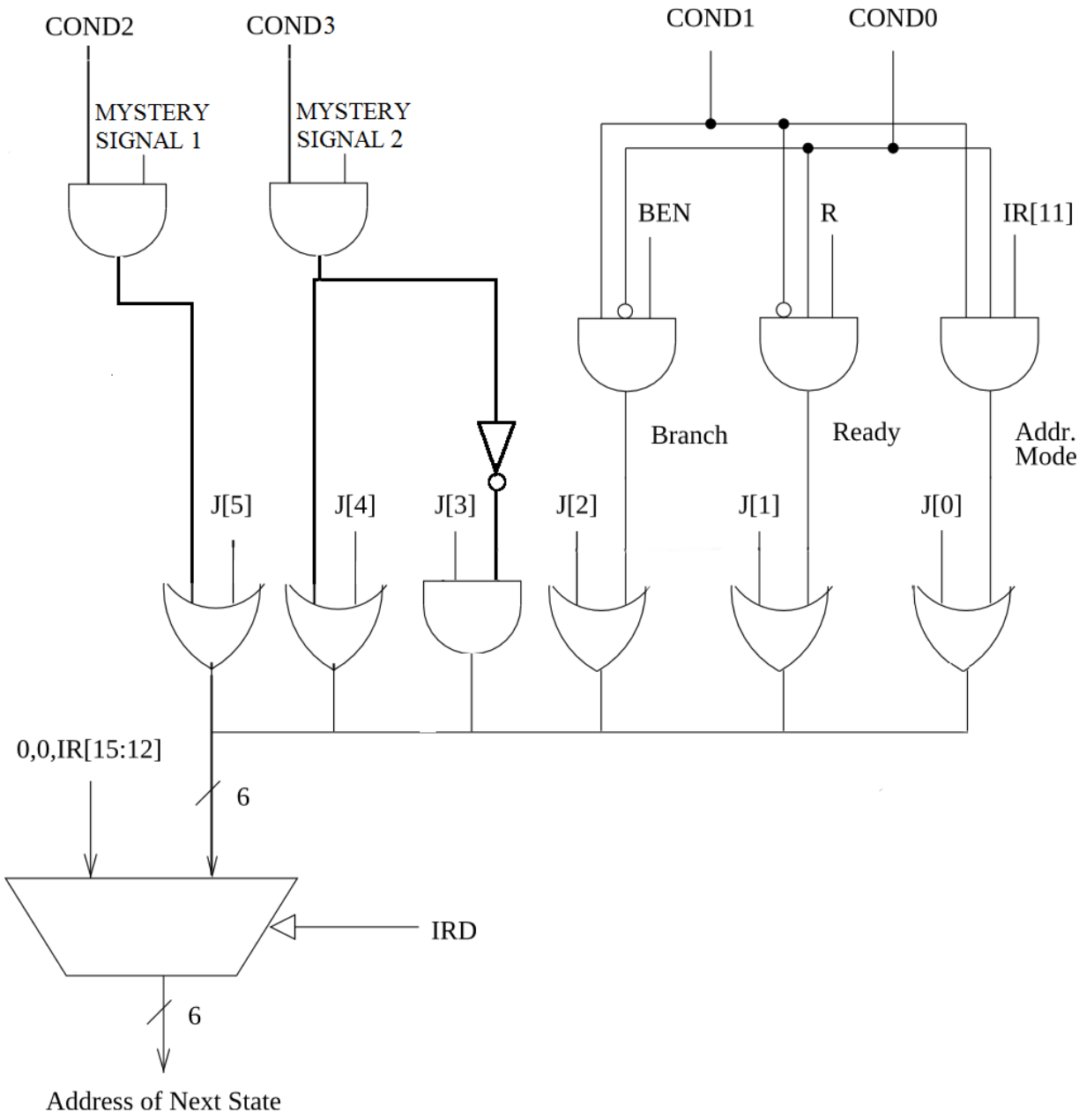
Describe what this instruction does. Show your work for partial credit.

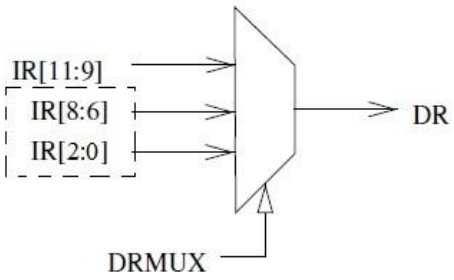
This instruction checks if the given string is a palindrome.

```
Code:
(char * sr1, *sr2;)
destR = 0;
while(sr1 < sr2){
    if (mem[sr1] != mem[sr2])
        return(fetch next instruction)
    sr1++;
    sr2--;
}
destR = 1;
return(fetch next instruction)
```

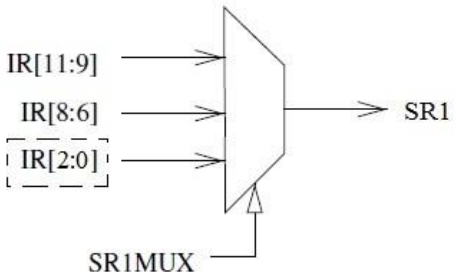
Initials:


GateMARMUX
GatePC
>= 0 ?
Mystery2
REG_IN MUX
LD.PC
PC
+2
PCMUX
MARMUX
LD.TEMP1
TEMP1
LD.TEMP2
TEMP2
REG FILE
LD.REG
DR
SR2
SR2 OUT
SR1 OUT
SR1
XOR
ZEXT & LSHF1
LSHF1
ADDR1MUX
ADDR2MUX
[7:0]
[10:0]
[8:0]
[5:0]
[4:0]
SEXT
SR2MUX
CONTROL
MYSTERY MUX
GateTEMP3
LD.IR
IR
Mystery1
= 0 ?
LD.CC
N Z P
LOGIC
ALUK
ALU
SHF
IR[5:0]
GateALU
GateSHF
GateMDR
MAR
LD.MAR
DATA.SIZE
MAR[0]
WE LOGIC
R.W
MIO.EN
INPUT
OUTPUT
KBDR
DDR
KBSR
DSR
WE1 WE0
MEMORY
MEM.EN
ADDR.CTL. LOGIC
MDR
LD.MDR
INMUX

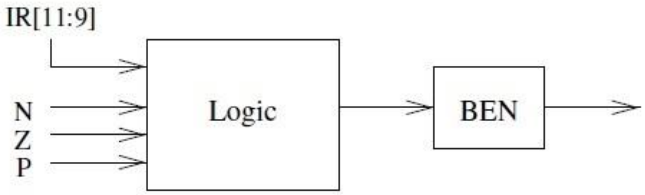

COND2
COND3
COND1
COND0
MYSTERY SIGNAL 1
MYSTERY SIGNAL 2
BEN
R
IR[11]
Branch
Ready
Addr. Mode
J[5]
J[4]
J[3]
J[2]
J[1]
J[0]
0,0,IR[15:12]
6
IRD
6
Address of Next State


IR[11:9]
IR[8:6]
IR[2:0]
DR
DRMUX

(a)

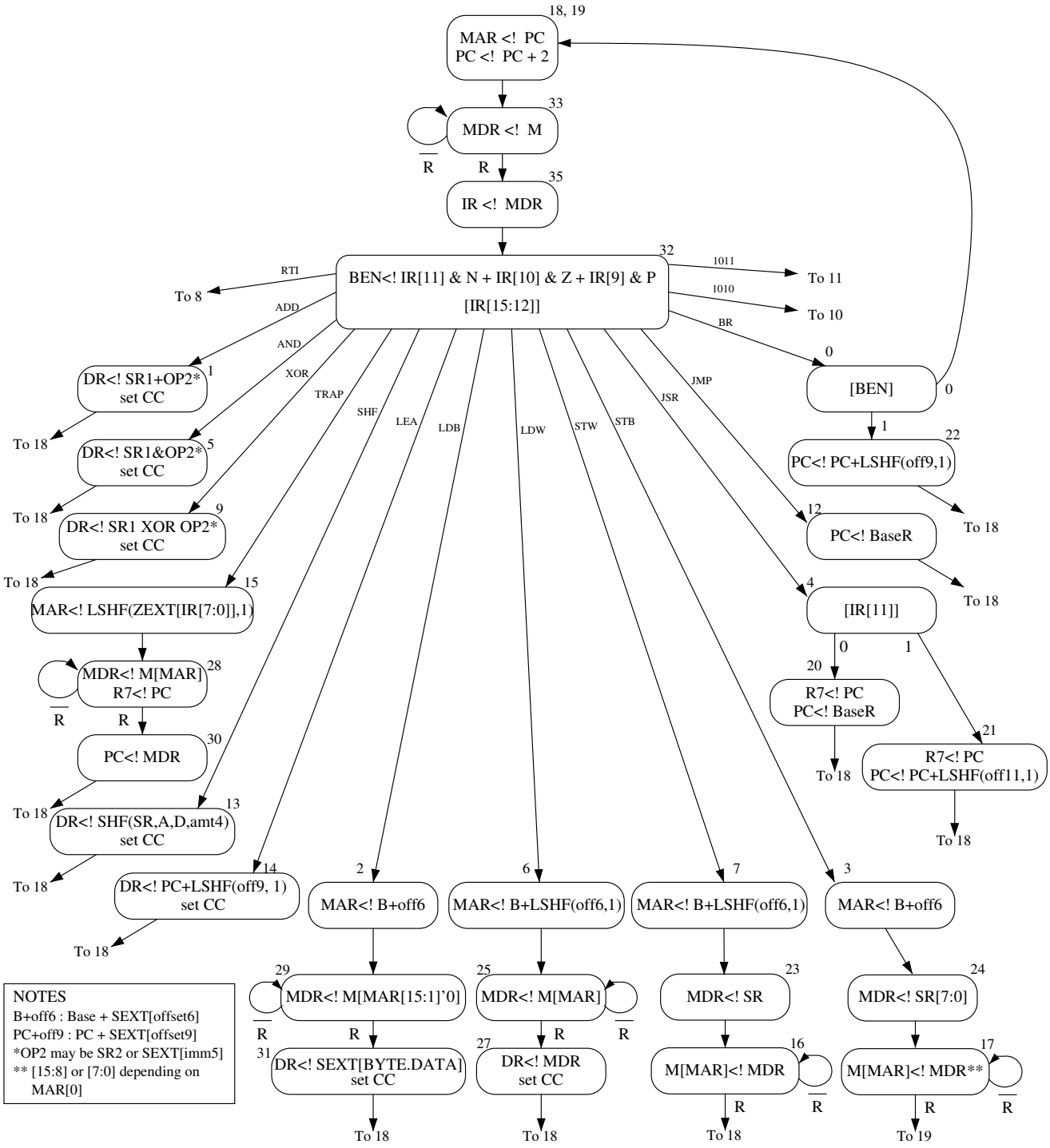

IR[11:9]
IR[8:6]
IR[2:0]
SR1
SR1MUX

(b)


IR[11:9]
N
Z
P
Logic
BEN

(c)

Initials:


18, 19
MAR <! PC
PC <! PC + 2
33
MDR <! M
R
R
35
IR <! MDR
32
BEN<! IR[11] & N + IR[10] & Z + IR[9] & P
[IR[15:12]]
RTI
To 8
ADD
AND
XOR
TRAP
SHF
LEA
LDB
LDW
STW
STB
JSR
JMP
BR
1011
To 11
1010
To 10
0
[BEN]
0
1
22
PC<! PC+LSHF(off9,1)
To 18
1
DR<! SR1+OP2*
set CC
To 18
5
DR<! SR1&OP2*
set CC
To 18
9
DR<! SR1 XOR OP2*
set CC
To 18
12
PC<! BaseR
To 18
15
MAR<! LSHF(ZEXT[IR[7:0]],1)
4
[IR[11]]
0
1
28
MDR<! M[MAR]
R7<! PC
R
R
20
R7<! PC
PC<! BaseR
To 18
21
R7<! PC
PC<! PC+LSHF(off11,1)
To 18
30
PC<! MDR
To 18
13
DR<! SHF(SR,A,D,amt4)
set CC
To 18
14
DR<! PC+LSHF(off9, 1)
set CC
To 18
2
MAR<! B+off6
6
MAR<! B+LSHF(off6,1)
7
MAR<! B+LSHF(off6,1)
3
MAR<! B+off6
29
MDR<! M[MAR[15:1]'0]
R
R
31
DR<! SEXT[BYTE.DATA]
set CC
To 18
25
MDR<! M[MAR]
R
R
27
DR<! MDR
set CC
To 18
23
MDR<! SR
16
M[MAR]<! MDR
R
R
To 18
24
MDR<! SR[7:0]
17
M[MAR]<! MDR**
R
R
To 19
NOTES
B+off6 : Base + SEXT[offset6]
PC+off9 : PC + SEXT[offset9]
*OP2 may be SR2 or SEXT[imm5]
** [15:8] or [7:0] depending on
MAR[0]

Stratchpad

VLIW Instruction	**ALU**	**MU**	**FPU**
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Initials:

VLIW Instruction	**ALU**	**MU**	**FPU**
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			

Initials:

VLIW Instruction	**ALU**	**MU**	**FPU**
1			
2			
3			
4			
5			
6			
7			
8			
9			
10			
11			
12			
13			
14			
15			
16			
17			
18			
19			
20			
21			
22			
23			
24			
25			