

CMU 18-447 INTRODUCTION TO COMPUTER ARCHITECTURE, SPRING 2014
HW 2: ISA TRADEOFFS, MICROPROGRAMMING AND PIPELINING

Instructor: Prof. Onur Mutlu

TAs: Rachata Ausavarungnirun, Varun Kohli, Xiao Bao Zhao, Paraj Tyle

Assigned: Wed., 1/29, 2014

Due: **Wed., 2/12, 2014 (Midnight)**

Handin: `/afs/ece/class/ece447/handin/hw2`

1 LC-3b Microcode [40 points]

We wrote the microcode for at least one state of the LC-3b microarchitecture in class. In this homework, you will complete the microcode for all states of the LC-3b. Refer to Appendix C of Patt and Patel for the LC-3b state machine and datapath and Appendix A of Patt and Patel for the LC-3b ISA description.

Fill out the microcode in the `microcode.csv` file handed out with this homework. Enter a 1 or a 0 or an X as appropriate for the microinstructions corresponding to states. You do not need to fill out states 8, 10 and 11. We fill out state 18 as an example. Please turn in this CSV file electronically along with your homework.

2 Data Reuse in Different ISAs (Optional)

Consider two machines, X and Y , described below.

Machine X: A three-address memory/memory machine that takes two memory locations as source operands and one memory location as a destination operand. Machine X 's ISA has the following characteristics:

- **OP M3, M1, M2:** Performs the binary operation OP on the values stored at memory locations $M1$ and $M2$ and stores the result back into memory location $M3$ ($M3 \leftarrow M1 \text{ OP } M2$).
- **OP M3, M1, Immediate:** Performs a binary operation OP on the value stored at memory location $M1$ and value **Immediate** and stores the result back into memory location $M3$ ($M3 \leftarrow M1 \text{ OP } \text{Immediate}$).
- Each instruction is encoded in 7 bytes.
- Data values are 4 bytes.

Machine Y: A three-address load/store machine whose sources and destination are registers. Values are loaded into registers using memory operations. Machine Y 's ISA has the following characteristics:

- **OP R3, R1, R2:** Performs a binary operation OP on the values stored at registers $R1$ and $R2$ and stores the result back into register $R3$ ($R3 \leftarrow R1 \text{ OP } R2$).
- **OP R3, R1, Immediate:** Performs a binary operation OP on the value stored at registers $R1$ and immediate value **Immediate** and stores the result back into register $R3$ ($R3 \leftarrow R1 \text{ OP } \text{Immediate}$).
- **LD R1, M:** Loads the value at memory location M into register $R1$.
- **ST R2, M:** Stores the value in register $R2$ into memory location M .
- There are 32 general-purpose registers.
- Each instruction is encoded in 4 bytes.
- Memory data values are 4 bytes.

Consider the following two programs:

Program A

```
A = A + 1;
B = B + 2;
C = C - 3;
D = D - 4;
```

Program B

```
B = B + A;
C = B + A;
D = A - C;
A = D - C;
```

- (a) For each of the two programs, *A* and *B*, write the corresponding assembly code for each of the two machines, *X* and *Y*. For Machine *Y*, ensure to (i) **reuse** the register values whenever possible and (ii) store the register values back into memory after executing all the code.
- (b) Compute the total number of bytes (instruction *and* data) transferred to/from memory for each of the two programs, *A* and *B*, for each of the two machines, *X* and *Y*.

	Code A	Code B
Machine X		
Machine Y		

- (c) Between machines *X* and *Y*, (i) does the machine that transfers the smaller total number of bytes for Program *A* also transfer the smaller total number of bytes for Program *B*? (ii) If yes, explain why this is the case; otherwise, explain why there is a difference.

3 Addressing Modes [12 points]

We covered the following addressing modes in class:

- Absolute
- Register indirect
- Based (base + displacement)
- Scale indexed (base + index \times constant)
- Memory indirect
- Auto increment/decrement (by 1 byte)

Consider the following high-level programs:

- ```
uint8_t a[100]; // a is allocated in memory
for (i = 0; i < 150; i++) {
 a[i] = 5;
}
```
- ```
int a[100]; // a is allocated in memory
for (i = 0; i < 150; i++) {
    a[i] = 5;
}
```
- ```
int *p; // *p is allocated in memory
*p = 150;
```

- `int **p; // *p and **p are allocated in memory`  
`**p = 150;`

Assume that in the first two programs, a register contains the address of the start of the array, and in the last two programs, a register contains the value of `p`.

For each of the above three programs, which of the addressing modes, do you think, would lead to the minimum number of instructions? (Note that no addressing mode fits perfectly. You might require other instructions for address computation.)

#### 4 Addressability (Optional)

Say we have 64 megabytes of storage. Calculate the number of bits required to address a location if:

- (i) the ISA is bit-addressable
  - (ii) the ISA is byte-addressable
  - (iii) the ISA is 8-byte-addressable
  - (iv) the ISA is 32-byte-addressable
- (b) A nibble is half of a byte (i.e., it contains 4 bits). Write an ARM program that computes the sum of two nibbles: (1) the nibble at the offset stored in `$4` of the word located at the address stored in `$5` and (2) the nibble at the offset stored in `$6` of the word located at the address stored in `$7`. You are only allowed to perform load/store accesses to addresses that are 4-byte (32-bit) aligned. Assume a little-endian format. The computed sum should be stored (zero-extended) in `$2`.

#### 5 Microarchitecture vs. ISA [15 points]

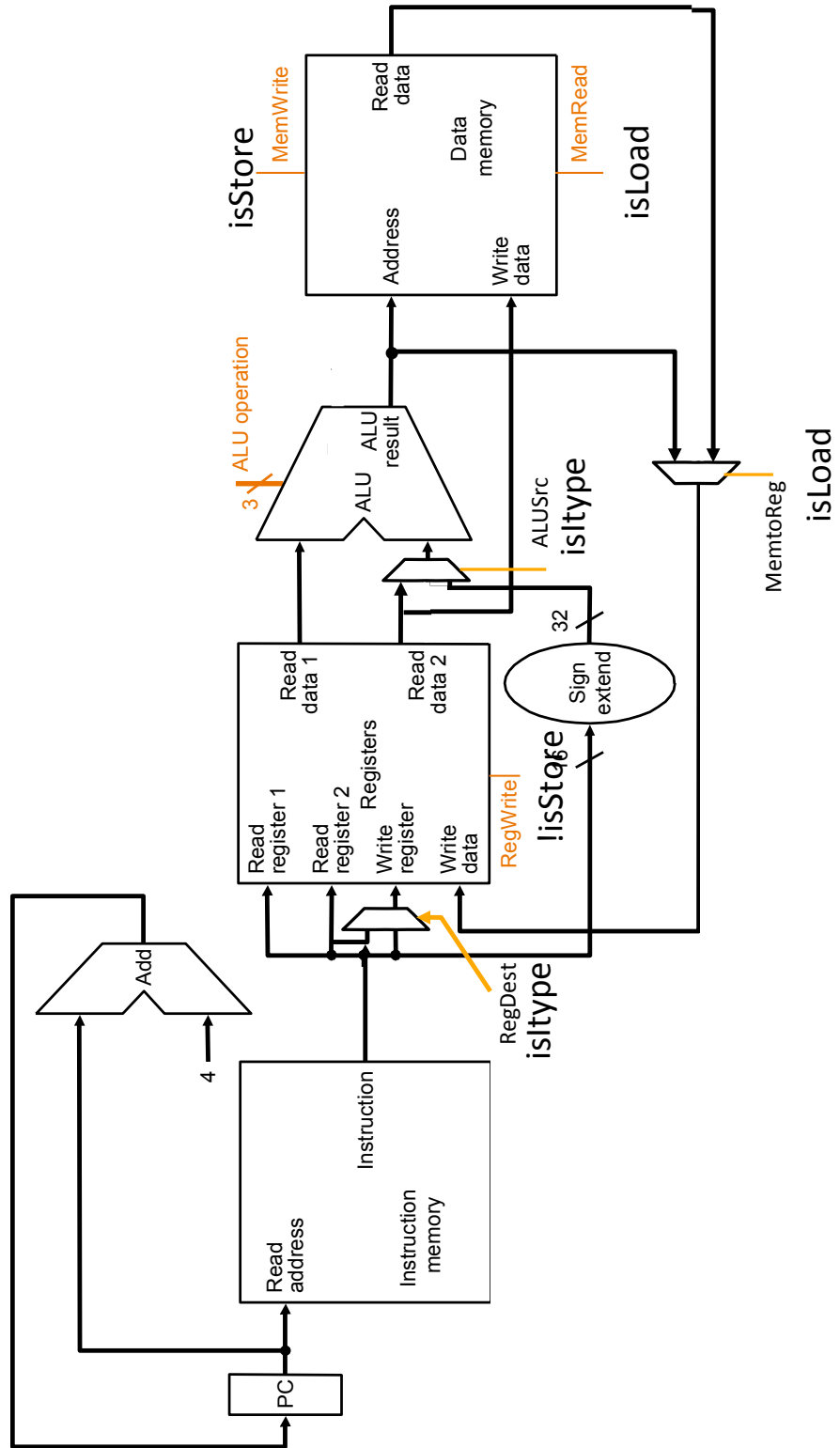
- Briefly explain the difference between the microarchitecture level and the ISA level in the transformation hierarchy. What information does the compiler need to know about the microarchitecture of the machine in order to compile the program correctly?
- Classify the following attributes of a machine as either a property of its microarchitecture or ISA:
  - The machine does not have a subtract instruction.
  - The ALU of the machine does not have a subtract unit.
  - The machine does not have condition codes.
  - A 5-bit immediate can be specified in an `ADD` instruction.
  - It takes  $n$  cycles to execute an `ADD` instruction.
  - There are 8 general purpose registers.
  - A 2-to-1 mux feeds one of the inputs to ALU.
  - The register file has one input port and two output ports.

#### 6 Single-Cycle Processor Datapath [30 points]

In this problem, you will modify the single-cycle datapath we built up in lecture to support the `JAL` instruction. The datapath that we will start with (from Lecture 5, Slide 53) has been reproduced on the next page. Your job is to implement the necessary data and control signals to support the `JAL` instruction, which we define to have the following semantics:

$$\begin{aligned} \text{JAL : } R31 &\leftarrow PC + 4 \\ PC &\leftarrow PC_{31..28} \parallel \text{Immediate} \parallel 0^2 \end{aligned}$$

Add to the datapath on the next page the necessary data and control signals to implement the JAL instruction. Draw and label all components and wires very clearly (give control signals meaningful names; if selecting a subset of bits from many, specify exactly which bits are selected; and so on).



## 7 Pipelining [30 points]

Given the following code:

```
MUL R3, R1, R2
ADD R5, R4, R3
ADD R6, R4, R1
MUL R7, R8, R9
ADD R4, R3, R7
MUL R10, R5, R6
```

Note: Each instruction is specied with the destination register rst.

Calculate the number of cycles it takes to execute the given code on the following models:

- (i) A non-pipelined machine
- (ii) A pipelined machine with scoreboarding and ve adders and ve multipliers without data forwarding
- (iii) A pipelined machine with scoreboarding and ve adders and ve multipliers with data forwarding.
- (iv) A pipelined machine with scoreboarding and one adder and one multiplier without data forwarding
- (v) A pipelined machine with scoreboarding and one adder and one multiplier with data forwarding

Note: For all machine models, use the basic instruction cycle as follows:

- Fetch (one clock cycle)
- Decode (one clock cycle)
- Execute (MUL takes 6, ADD takes 4 clock cycles). The multiplier and the adder are not pipelined.
- Write-back (one clock cycle)

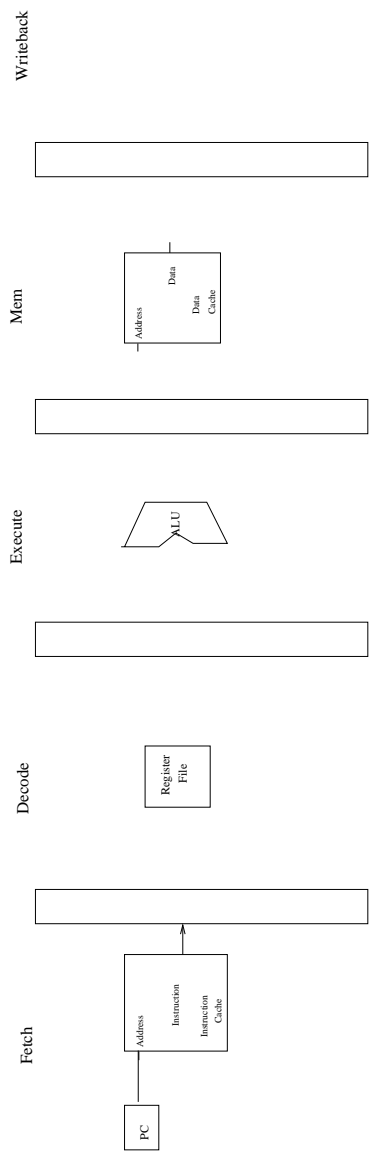
Do not forget to list any assumptions you make about the pipeline structure (e.g., how is data forwarding done between pipeline stages)

## 8 Fine Grain Multi-threading [40 points]

Consider a design “Machine I” with five pipeline stages: fetch, decode, execute, memory and writeback. Each stage takes 1 cycle. The instruction and data caches have 100% hit rates (i.e., there is never a stall for a cache miss). Branch directions and targets are resolved in the execute stage. The pipeline stalls when a branch is fetched, until the branch is resolved. Dependency check logic is implemented in the decode stage to detect flow dependences. The pipeline does not have any forwarding paths, so it must stall on detection of a flow dependence.

In order to avoid these stalls, we will consider modifying Machine I to use fine-grained multithreading.

- (a) In the five stage pipeline of Machine I shown below, clearly show what blocks you would need to add in each stage of the pipeline, to implement fine-grained multithreading. You can replicate any of the blocks and add muxes. You don’t need to implement the mux control logic (although provide an intuitive name for the mux control signal, when applicable).



- (b) The machine's designer first focuses on the branch stalls, and decides to use fine-grained multithreading to keep the pipeline busy no matter how many branch stalls occur. What is the minimum number of threads required to achieve this? Why?
- (c) The machine's designer now decides to eliminate dependency-check logic and remove the need for flow-dependence stalls (while still avoiding branch stalls). How many threads are needed to ensure that no flow dependence ever occurs in the pipeline? Why?

A rival designer is impressed by the throughput improvements and the reduction in complexity that FGMT brought to Machine I. This designer decides to implement FGMT on another machine, Machine II. Machine II is a pipelined machine with the following stages.

|           |                                                                            |
|-----------|----------------------------------------------------------------------------|
| Fetch     | 1 stage                                                                    |
| Decode    | 1 stage                                                                    |
| Execute   | 8 stages (branch direction/target are resolved in the first execute stage) |
| Memory    | 2 stages                                                                   |
| Writeback | 1 stage                                                                    |

Assume everything else in Machine II is the same as in Machine I.

- (d) Is the number of threads required to eliminate branch-related stalls in Machine II the same as in Machine I?

**YES NO** (Circle one)

If yes, why? If no, how many threads are required?

- (e) What is the minimum CPI (i.e., maximum performance) of each thread in Machine II when this minimum number of threads is used?
- (f) Now consider flow-dependence stalls. Does Machine II require the same minimum number of threads as Machine I to avoid the need for flow-dependence stalls?

**YES NO** (Circle one)

If yes, why? If no, how many threads are required?

- (g) What is the minimum CPI of each thread when this number of threads (to cover flow-dependence stalls) is used?
- (h) After implementing fine grained multithreading, the designer of Machine II optimizes the design and compares the pipeline throughput of the original Machine II (without FGMT) and the modified Machine II (with FGMT) both machines operating at their maximum possible frequency, for several code sequences. On a particular sequence that has no flow dependences, the designer is surprised to see that the new Machine II (with FGMT) has lower overall throughput (number of instructions retired by the pipeline per second) than the old Machine II (with no FGMT). Why could this be? Explain concretely.

## 9 Branch Prediction and Dual Path Execution [35 points]

Assume a machine with a 7-stage pipeline. Assume that branches are resolved in the sixth stage. Assume that 20% of instructions are branches.

- (a) How many instructions of wasted work are there per branch misprediction on this machine?
- (b) Assume  $N$  instructions are on the correct path of a program and assume a branch predictor accuracy of  $A$ . Write the equation for the number of instructions that are fetched on this machine in terms of  $N$  and  $A$ . (Please show your work for full credit.)
- (c) Let's say we modified the machine so that it used *dual path execution* like we discussed in class (where an equal number of instructions are fetched from each of the two branch paths). Assume branches are



resolved before new branches are fetched. Write how many instructions would be fetched in this case, as a function of  $N$ . (Please show your work for full credit.)

- (d) Now let's say that the machine combines branch prediction *and* dual path execution in the following way:

A branch confidence estimator, like we discussed in class, is used to gauge how confident the machine is of the prediction made for a branch. When confidence in a prediction is high, the branch predictor's prediction is used to fetch the next instruction; When confidence in a prediction is low, dual path execution is used instead.

Assume that the confidence estimator estimates a fraction  $C$  of the branch predictions have high confidence, and that the probability that the confidence estimator is wrong in its high confidence estimation is  $M$ .

Write how many instructions would be fetched in this case, as a function of  $N$ ,  $A$ ,  $C$ , and  $M$ . (Please show your work for full credit.)

## 10 Mysterious Instruction [40 points]

An engineer implemented the mystery instruction described below on the LC-3b. Unfortunately, we do not know what this engineer was thinking, and we can't figure out what the instruction does. It is your job to determine this. The mystery instruction is encoded as:

|      |    |    |    |    |    |   |     |   |   |   |   |   |   |   |   |
|------|----|----|----|----|----|---|-----|---|---|---|---|---|---|---|---|
| 15   | 14 | 13 | 12 | 11 | 10 | 9 | 8   | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| 1010 |    |    |    | DR |    |   | SR1 |   |   | 1 | 0 | 1 | 1 | 1 | 1 |

The instruction is only defined if the value of SR1 is not equal to zero.

The modifications we make to the LC-3b datapath and the microsequencer are highlighted in the attached figures (see the next three pages). We also provide the original LC-3b state diagram, in case you need it.

The additional control signals are

**GateLSHF/1:** NO, YES

**LD.TEMP/1:** NO, LOAD

**ALUK/3:** AND, ADD, XOR, PASSA, PASSB, DECA (Decrement A)

**COND/3:**

$COND_0$  ;Unconditional

$COND_1$  ;Memory Ready

$COND_2$  ;Branch

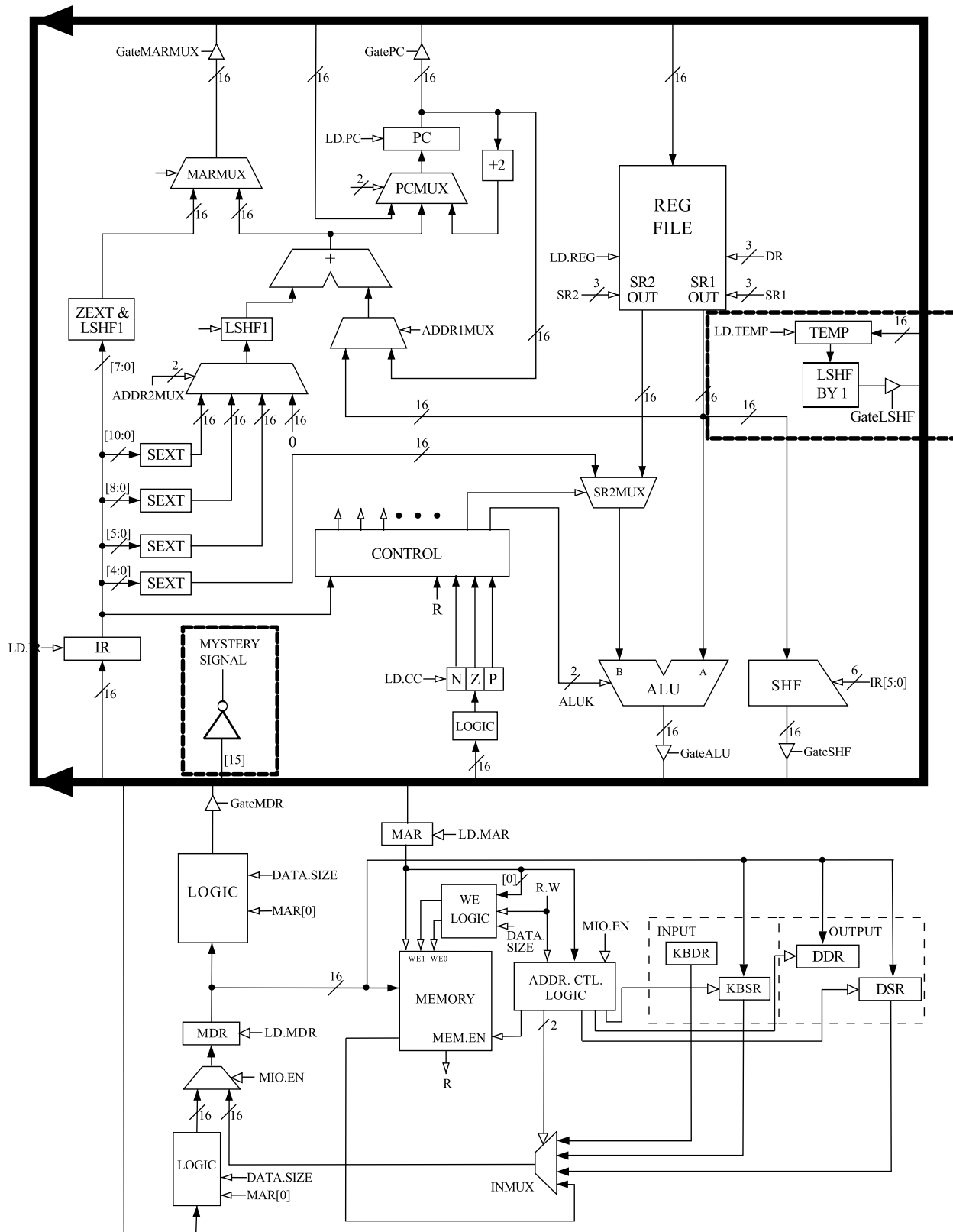
$COND_3$  ;Addressing mode

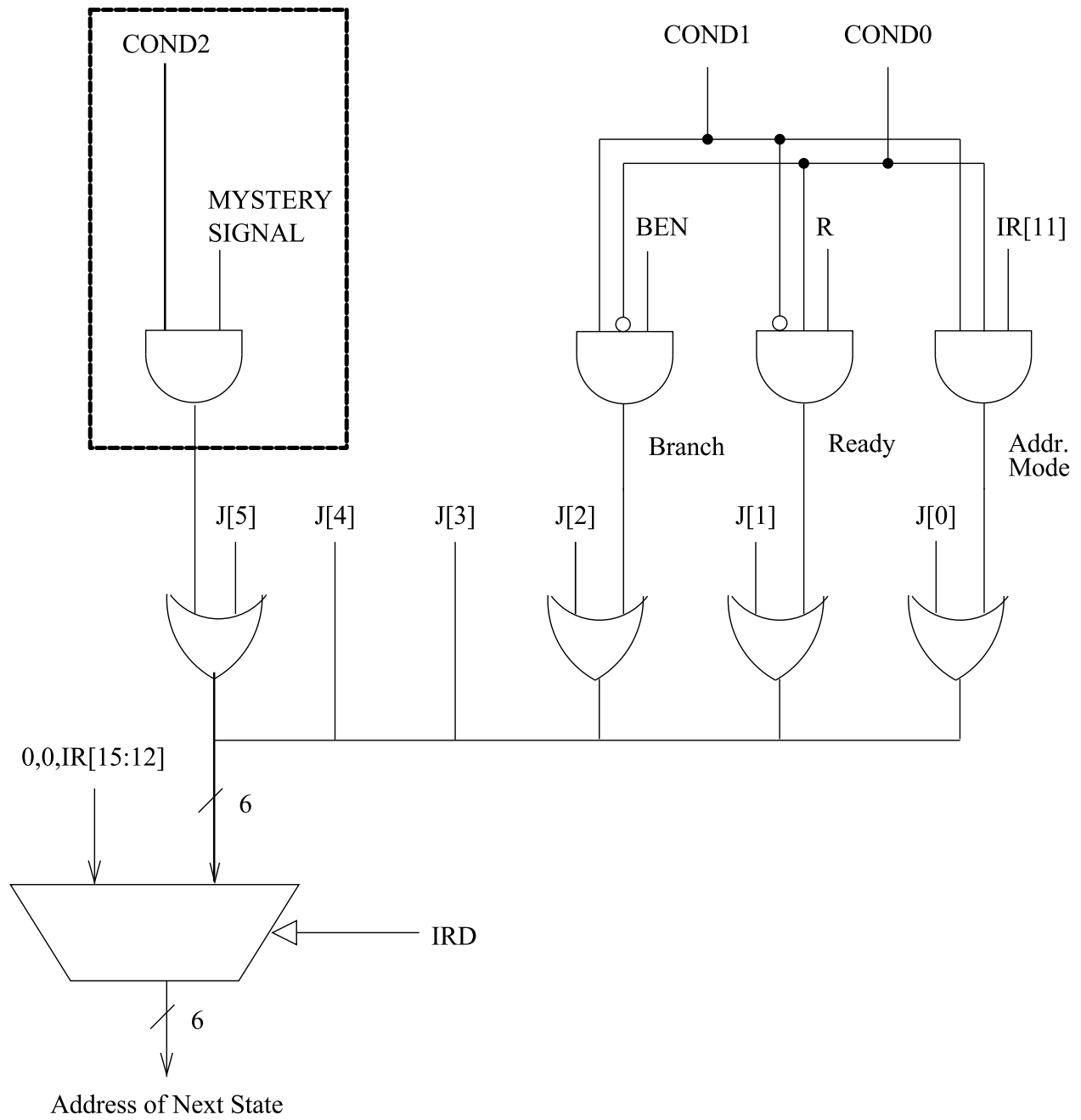
$COND_4$  ;Mystery

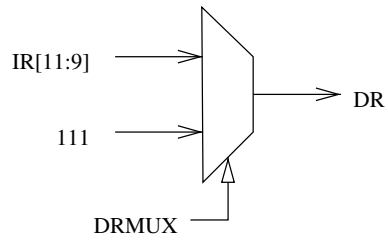
The microcode for the instruction is given in the table below.

| State       | Cond  | J      | Asserted Signals                                                 |
|-------------|-------|--------|------------------------------------------------------------------|
| 001010 (10) | COND0 | 101000 | ALUK = PASSB, GateALU, LD.REG, DRMUX = DR (IR[11:9])             |
| 101000 (40) | COND4 | 010010 | ALUK = PASSA, GateALU, LD.TEMP, SR1MUX = SR1 (IR[8:6])           |
| 110010 (50) | COND0 | 110011 | ALUK = DECA, GateALU, LD.REG, SR1MUX = DR, DRMUX = DR (IR[11:9]) |
| 110011 (51) | COND4 | 010010 | GateLSHF, LD.TEMP                                                |

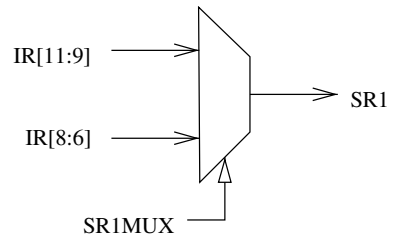
Describe what this instruction does.



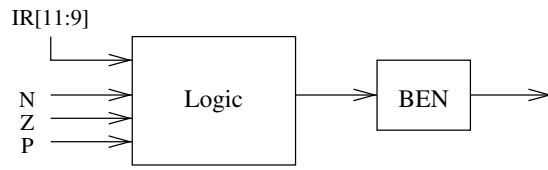




(a)



(b)



(c)

