Department of Electrical and Computer Engineering
The University of Texas at Austin

EE 360N, Fall 2004
Yale Patt, Instructor
Aater Suleman, Huzefa Sanjeliwala, Dam Sunwoo, TAs
Final Exam, December 13, 2004

Name **(1 point)**:__________________________

Problem 1 (24 points):_______

Problem 2 (10 points):_______

Problem 3 (15 points):_______

Problem 4 (20 points):_______

Problem 5 (25 points):_______

Problem 6 (25 points):_______

Total (120 points):______________

Note: Please be sure that your answers to all questions (and all supporting work that is required) are contained in the space provided.

Note: Please be sure your name is recorded on each sheet of the exam.

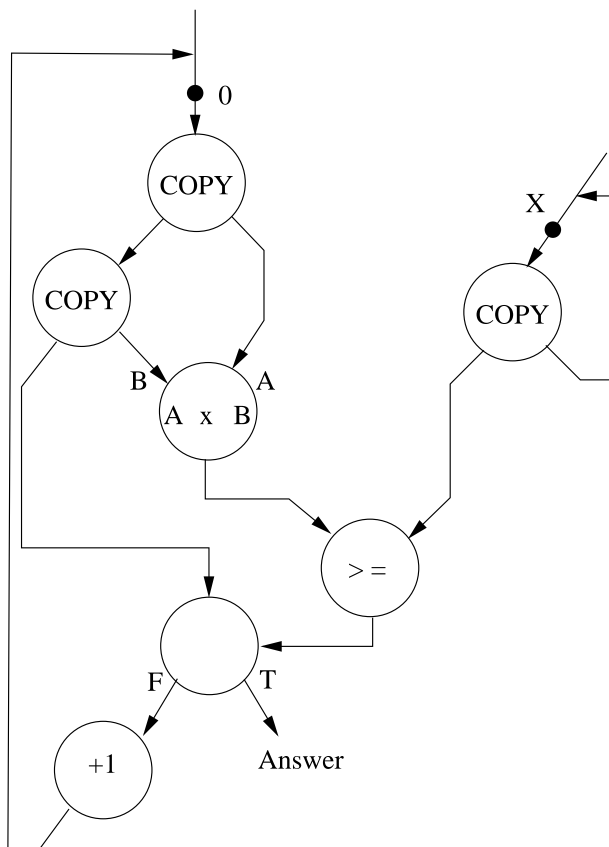
GOOD LUCK!

Name:________________________________

Problem 1 (24 points)

Part a (8 points): Ahmdahl's good friend Lehgdahl, recognizing that it is really not the case that all parts of the code run with full parallelism or no parallelism, has come up with a new law. Lehgdahl believes that code runs with half the processing elements busy, all the processing elements busy, or sequential. Let p be the number of processors available, α be the fraction of the program that can keep all p processors busy, and β the fraction of the program that can keep half of the processors busy.
Derive a formula for Speedup as a function of p, α, and β.

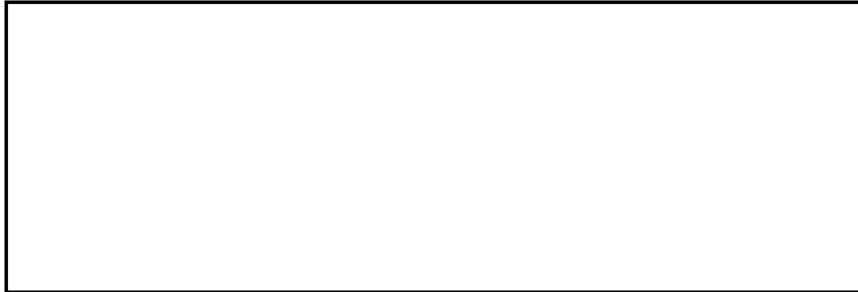
Part b (8 points): If X is a positive integer, what does the dataflow graph below compute?



ANSWER:

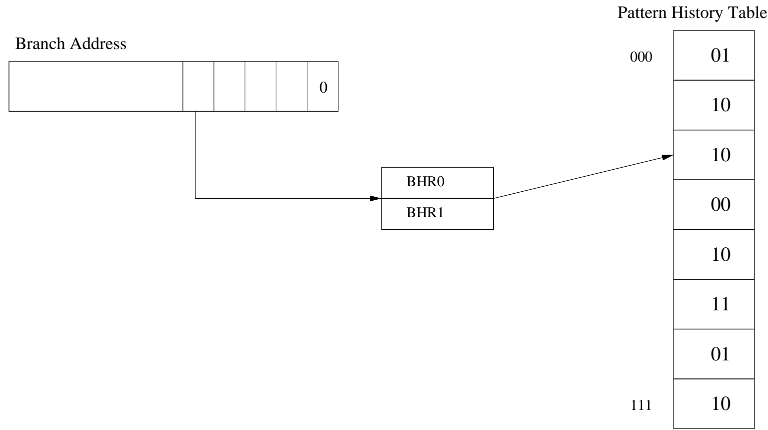
Name:__________________________

Part c (8 points): Goodman's cache coherency scheme requires all caches invalidate a block if they see a bus write. This would not be necessary if the cache generating the bus write instead supplied to all the other caches the data it wrote, so they could update their respective blocks. Memory, of course, doesn't update because that would be too slow. Thus, at any point in time, consistency of the data is maintained, even though memory is out-of-date. Cache misses can be supplied by caches on the bus who have the data. Only if no one has the data does memory service the cache miss. Suppose one copy of a particular "dirty" block is selected for replacement (i.e., a victim). It does not have to be written back to memory since other caches have the block and can supply it if another processor subsequently requests it. The above scheme has a tragic flaw which will prevent correct operation of the multiprocessor system? What is it?



Name:________________________________

Problem 2 (10 points): We show below a two-level branch predictor that we plan to use in the pipelined version of the LC-3b. There are exactly two Branch History Registers (BHR0 and BHR1) and one Pattern History Table (PHT). Bit 4 of the address of the branch instruction is used to select one of the two 3-bit BHRs. The contents of the selected BHR is used as an index to the table of saturating 2-bit counters to obtain the prediction.



Part a: Which variation of the two-level predictor is this? Circle the correct answer:

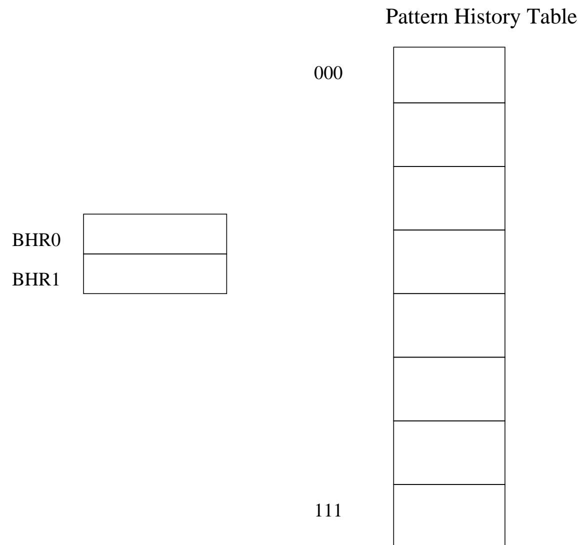
GAg, GAs, GAp, SAg, SAs, SAp, PAg, PAs, PAp.

Part b: Assume BHR0 contains 010, BHR1 contains 100 (the most recent branch corresponds to the most significant bit), and the counters in the PHT are as shown in the figure above.

The branch instruction at address x3010 is fetched. The two-level predictor predicts (circle one):

Taken Not-taken

After the branch is resolved, we learn it was taken. Assume no other branches have been encountered (BAD assumption, but for today, we will let it slide). Update the data structures accordingly.



Name:____________________________________

Problem 3 (15 points): The largest positive normalized value that can be exactly represented using a k-bit floating point number is 2016. The smallest positive subnormal number that can be exactly represented in this format is 1/256.

Note: All the floating point representations are based on the IEEE floating point standard.

Part a: How many bits are used for sign, exponent, and fraction in this format? Show your work.

Sign [ ] Exponent [ ] Fraction [ ]

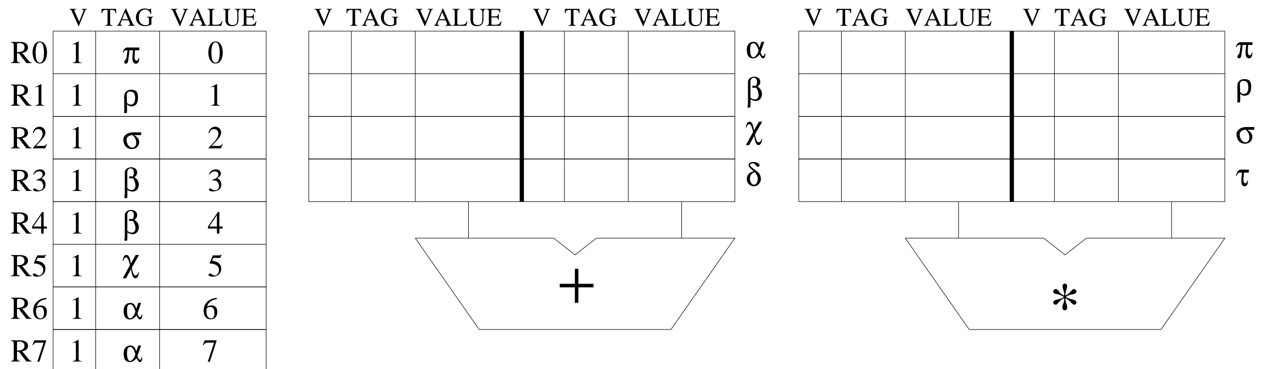
Part b: What is the excess (BIAS) for the excess code:

[ ]

Name:____________________________________

Problem 4 (20 points):

Initially, the register file of the Tomasulo-like machine shown below contains valid data in all registers and the reservation stations are empty.

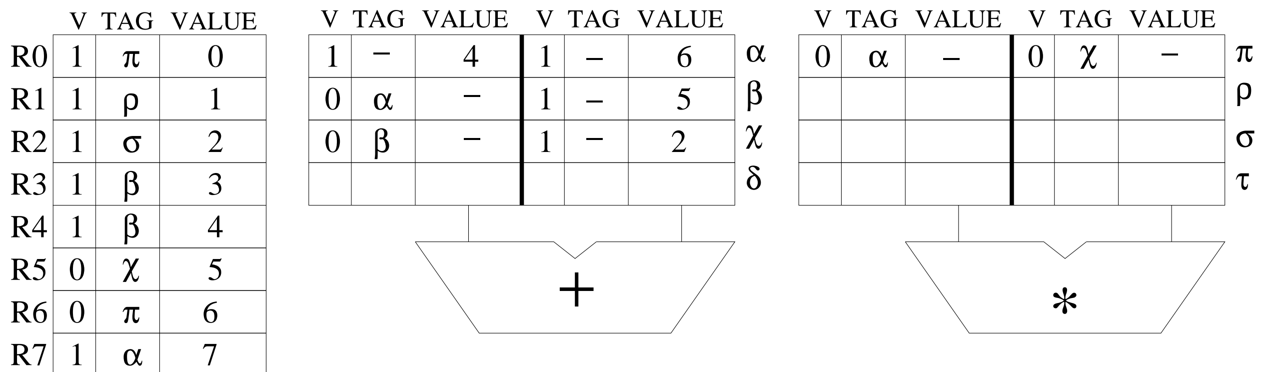


	V	TAG	VALUE
R0	1	π	0
R1	1	ρ	1
R2	1	σ	2
R3	1	β	3
R4	1	β	4
R5	1	χ	5
R6	1	α	6
R7	1	α	7

V	TAG	VALUE	V	TAG	VALUE	
						α
						β
						χ
						δ

V	TAG	VALUE	V	TAG	VALUE	
						π
						ρ
						σ
						τ

Four instructions are then fetched, decoded and issued in program order, none are executed. At that point the register file and reservation stations are as shown below:



	V	TAG	VALUE
R0	1	π	0
R1	1	ρ	1
R2	1	σ	2
R3	1	β	3
R4	1	β	4
R5	0	χ	5
R6	0	π	6
R7	1	α	7

V	TAG	VALUE	V	TAG	VALUE	
1	–	4	1	–	6	α
0	α	–	1	–	5	β
0	β	–	1	–	2	χ
						δ

V	TAG	VALUE	V	TAG	VALUE	
0	α	–	0	χ	–	π
						ρ
						σ
						τ

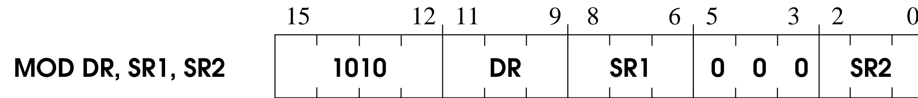
Using only instructions of the form `ADD Ri, Rj, Rk` and `MUL Ri, Rj, Rk`, where `Ri` is the destination register and `Rj` and `Rk` are the source registers, show the four instructions in program order.

1	
2	
3	
4	

Name:__________________________

Problem 5 (25 points):

We wish to add to the LC-3b the new instruction MOD, which calculates SR1 modulo SR2 and stores the result in DR. Recall A mod B is the remainder when integer A is divided by integer B. We make the assumption that both SR1 and SR2 contain positive numbers (non-zero). We will use the unused opcode 1010 for this purpose. The format of the instruction will be



Example:
After the instruction MOD R1,R2,R3 is executed, where R2 contains x0010, R3 contains x0005, R1 will contain the value x0001.

All the modifications to the datapath necessary to support MOD are shown on the next page. You are required to use **this** datapath as is.

Name:__________________________

Problem 5 continued:

To implement MOD, we have made the following changes to the LC-3b data path:

1. A three-input MUX to the A input of the ALU.
This requires a new control field, AMUX, specified as follows:
    SR1/00, the SR1 source from the original data path
    A/01, a constant value A
    B/10, a constant value B

2. The DRMUX now has 3 inputs.
    IR[11:9]/00, same as in original data path
    111/01, R7, same as in original data path
    IR[2:0]/10, SR2

3. We have also added a BENMUX at the input of the BEN register, as shown.
This requires a new control signal, BENMUX, specified as follows:
    Branch Decision Logic/0
    BUS[15]/1

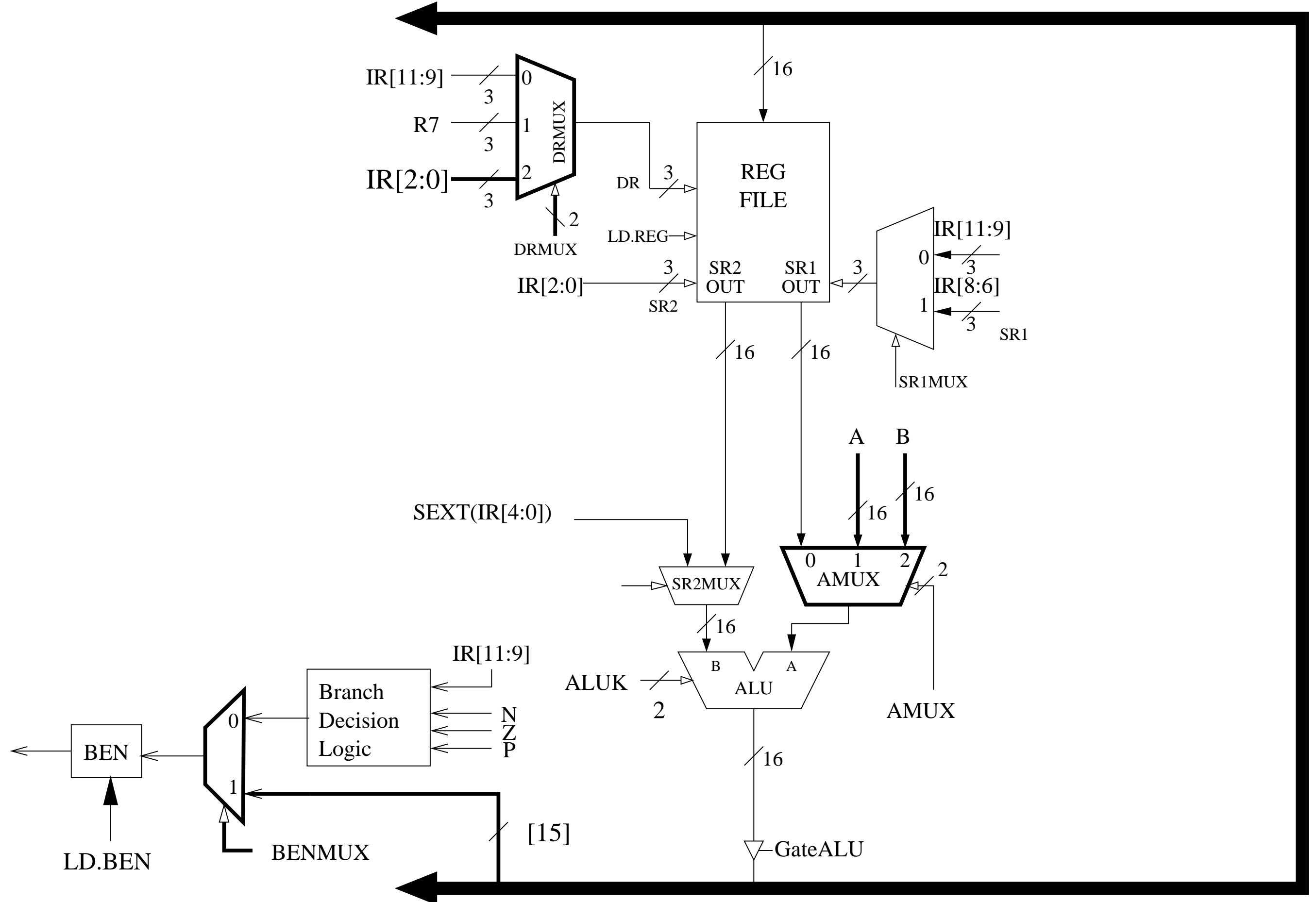


Figure 1: Modified datapath to support MOD instruction

Name:__________________________

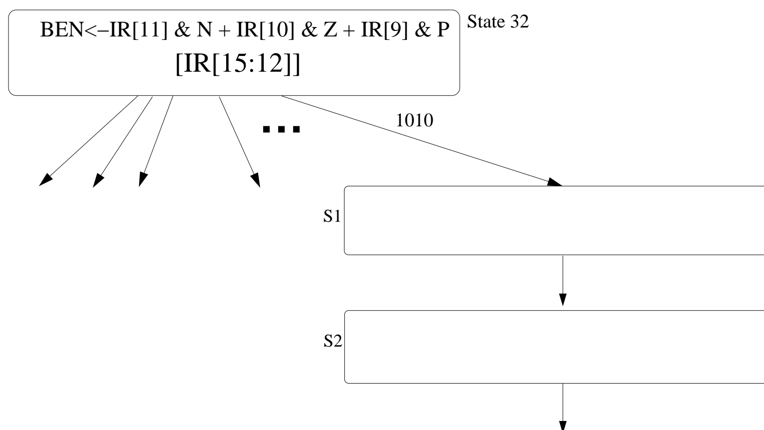
Problem 5 continued:

Part a. (4 points): Identify the 16-bit constants A and B

A:

B:

Part b. (12 points): We show below the beginning of the state diagram necessary to implement MOD. Using the notation of the LC-3b State Diagram, add the bubbles you need to implement the MOD instruction. Describe inside each bubble what happens in each state. You should be able to implement this in fewer than 12 states. (A TA found a solution that required only 8 states).
Hint : If you clobber any register, do not forget to restore it.
Note : Do not use any memory operations.



Name:___________________________

Problem 5 continued:

Part c. (9 points): The processing in each state you just added is controlled by asserting or negating each control signal. Enter a 1 or a 0 as appropriate for the microinstructions corresponding to the states you have added. For ALUK signal, you can use one of the following values: ADD, AND, XOR, PASSA.

	LD.REG	AMUX[1:0]	SR1MUX	DRMUX[1:0]	ALUK[1:0]	BENMUX	LD.BEN	GateALU
S1								
S2								
S3								
S4								
S5								
S6								
S7								
S8								
S9								
S10								
S11								
S12								

Name:________________________

Problem 6. (25 points) Recall the pipelined LC-3b you implemented in Lab 5. The dependency check logic in DE stage stalled the pipeline whenever it detected a flow dependency. To reduce both the number stalls and the length of each stall, we can use *data forwarding*. Data forwarding uses additional hardware to provide the required operands directly from the inter-stage pipeline latches to the inputs of the functional units, instead of waiting for the register value to get updated in the final stage.

Consider the execution of the following instructions(Destination comes first for all instructions):

```
I1:  ADD R3, R2, R1
I2:  XOR R4, R3, R5
```

Instead of stalling I2 till R3 gets updated, we can use the value of R1+R2, computed in the AGEX stage, from the MEM latches.

With data forwarding, we only need to stall in one case. The stall occurs when we have two consecutive instructions, I1 and I2, where I2 depends on I1, and I1 is a load instruction. In this case, the data from memory is available only at the end of the MEM stage. So we stall for a cycle to get the data forwarded from the SR stage instead of the MEM stage.

Since your TAs were too busy, we hired an *Aggie* to design the logic to support data forwarding for the registers. The changes made to the pipelined datapath are highlighted in the figures. (Refer to **SEPARATE** handout) The C code that implements the logic for the dependency stall signal and the forwarding MUX select signal is shown below.
Note: For simplicity, we do not implement forwarding for the condition codes. (CC dependency logic remains unchanged.)

```
/*  REGISTER DEPENDENCY STALL LOGIC
 *  Similar to what you implemented in Lab 5 except that
 *  you only stall when there is a load instruction in the AGEX stage
 *  on which the instruction in DE stage depends.
 *  NOTE: AGEX_isLOAD is a new control signal that is set
 *       when there is a load instruction in AGEX.
 */
if(V_AGEX_LD_REG && AGEX_isLOAD && DE_V)
     if ( (AGEX_DRID==SR1_ID) && SR1_NEEDED) ||
          (AGEX_DRID==SR2_ID) && SR2_NEEDED)    )
          REG_DEP_STALL = 1;
     else REG_DEP_STALL = 0;
else REG_DEP_STALL = 0;
```

```
/*  FORWARDING MUX SELECT LOGIC for SR1
 *  Similar logic for SR2
 */
if(V_AGEX_LD_REG && ( (AGEX_DRID == SR1_ID) && SR1_NEEDED ) )
     FWD_SR1_MUX_SEL = 1;       /* Forward from MEM latches */
else if (V_MEM_LD_REG && ( (MEM_DRID == SR1_ID) && SR1_NEEDED ) )
     FWD_SR1_MUX_SEL = 2;       /* Forward from SR latches */
else FWD_SR1_MUX_SEL = 0;       /* Read from register file */
```

Name:_______________________

Problem 6. (Continued)

We reviewed the design and found that this implementation does **NOT** work in all cases.

Given below are 2 pieces of code. Dependencies are highlighted in bold.
Your job: For each piece of code, first identify whether the proposed design works correctly or not. In the space provided below: if it works correctly, specify the **number of cycles** it takes to execute the code and the number of cycles it would take if we had not added forwarding. If it doesn't work, explain why it doesn't work(in less than 25 words). (Assume that all accesses to memory are hits)

Part a

```
LDW R2, R1, #0
ADD R3, R2, R4
ADD R5, R3, R2
AND R6, R6, R7
```

Circle one: WORKS / DOESN'T WORK

Part b

```
XOR R4, R1, R5
AND R6, R3, R2
ADD R7, R4, R0
XOR R2, R4, R6
```

Circle one: WORKS / DOESN'T WORK

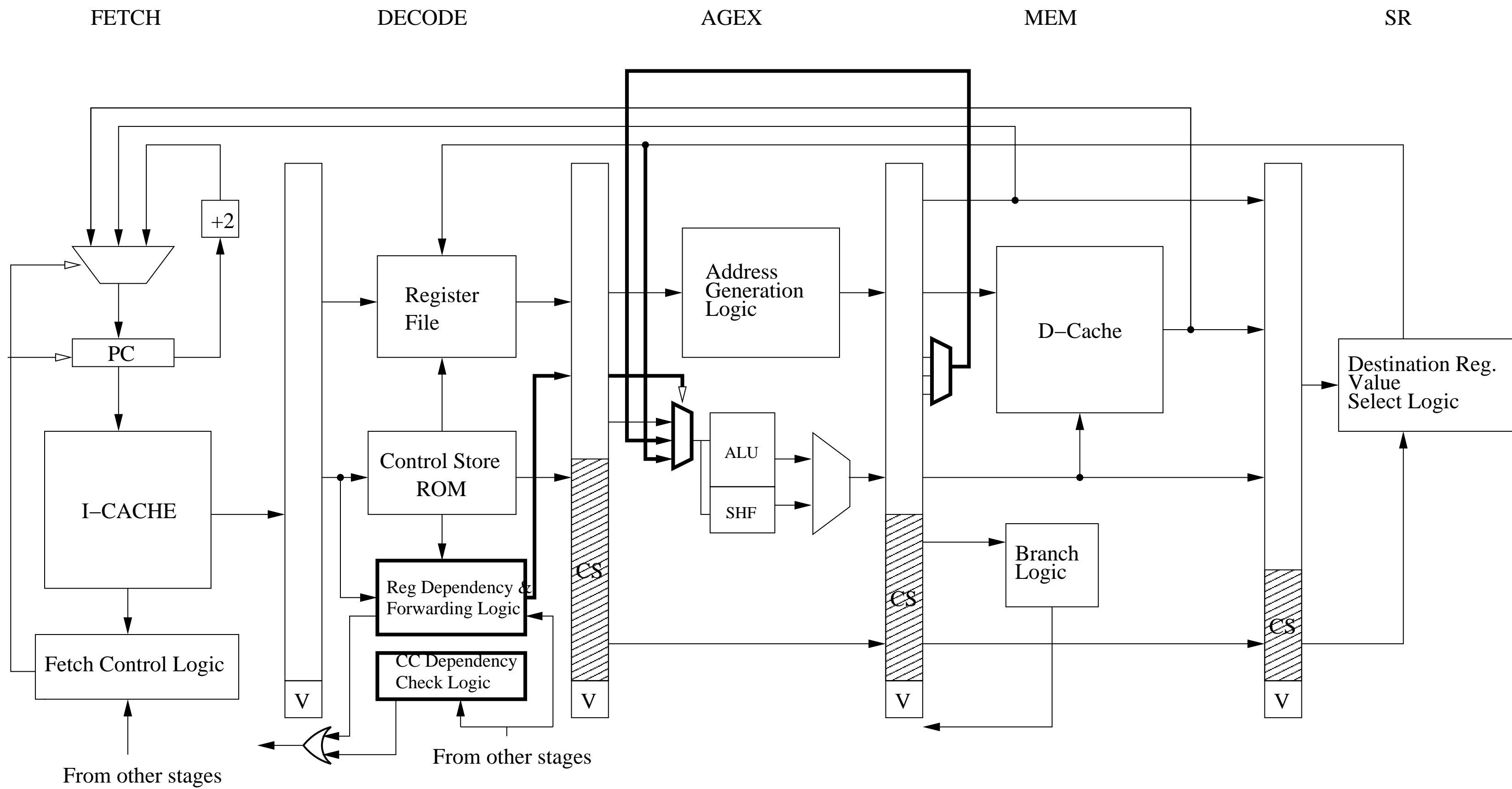


Fig.1 LC−3b pipeline diagram