CPU Control Design

ALU + Regfile + Memory + PC + Control FSM

Which addressing modes should you implement?

- Read the ISA.pdf, as well as the CR16 programmer reference manual on Canvas
- Some addressing modes we have to implement, others are your choice
- Register mode (R-type instructions): compulsory
    - ADD Rsrc Rdest: $R_{dest} \leftarrow R_{src} + R_{dest}$
    - Arithmetic, logical, Move: are R-type instructions
    - Make a list of all these instructions

Which addressing modes should you implement?

- Immediate mode (I-type instructions): compulsory
  - ADDI Imm Rdest: $R_{dest} \leftarrow \$Imm + R_{dest}$
  - Arithmetic, logical, Move: are included in I-type instructions
  - ISA.pdf shows: ADDI, SUBI, CMPI, ANDI, XORI,…, MOVI
  - They behave similarly as R-type instructions
  - Make a list of all these instructions

Which addressing modes should you implement?

- Direct/Absolute Addressing mode (Dir-type instructions): Not compulsory, can do without
  - ADD Rdest, [addr] : $R_{dest} \leftarrow R_{dest} + [mem\ \ addr]$
  - [mem addr] = data resides in memory, whose address is "mem addr"
  - Not implemented in CR16
  - More complicated, requires memory access

Which addressing modes should you implement?

- Indirect Addressing mode (Ind-type instructions): Compulsory
  - LOAD Rdest, Raddr : $R_{dest} \leftarrow [R_{addr}]$
  - Load data into Rdest, where data resides in memory whose address in stored in Raddr register
  - STOR Rsrc, Raddr: $[Raddr] \leftarrow Rsrc$
  - Store the data of Rsrc into memory at address [Raddr]
  - We will build a Load-Store machine: Use LOAD/STOR to fetch data into regfile, and then perform R-type instructions for computations!

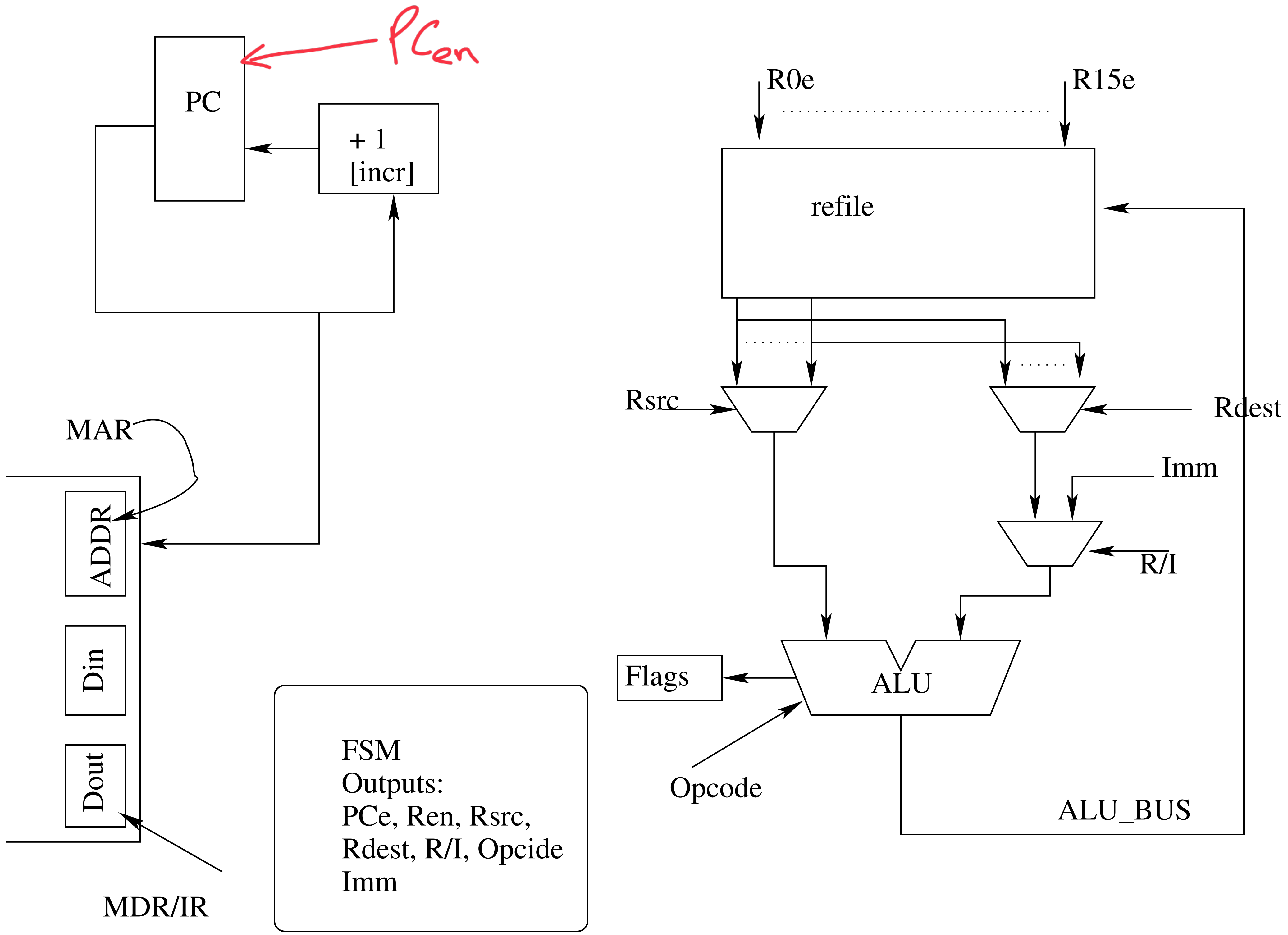
Which addressing modes should you implement?

- PC-Relative/Displacement Addressing mode (Rel-type instructions): Compulsory
  - Conditional Branches and Jumps: *e.g., "Bcond disp":*
  - If condition cond is met, branch to memory address (PC + disp)
  - PC <= PC + disp
  - ISA.pdf: *disp* = 8-bit 2's complement integer (bit vector), disp is given in the opcode
  - Branch versus Jump: "Jcond R_{target}": Jump to address that is stored in R_{target}
    - If condition is met, then PC = R_{target}
  - Condition codes are given on the page 6 in ISA.pdf. You have to implement a few, EQ, NE: Zero flags, CS, CC: Carry, GT, LE: Negative, FS, FC: Overflow

Approach

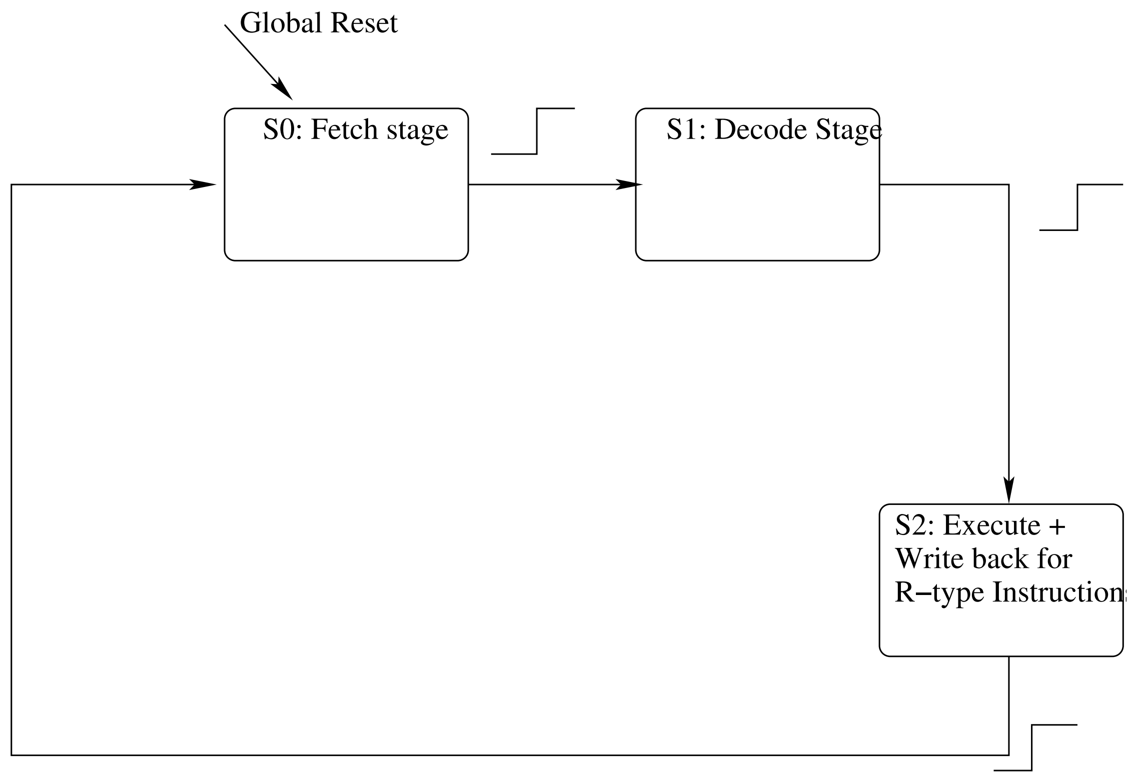
- First you should implement R-type instructions
  - Design Datapath to support R-type instructions: Program Counter [PC = PC + 1]
  - Design FSM
- Then implement Load/Store instructions
  - Augment the Datapath: [PC = Raddr]
  - Augment your FSM for Load/Store Instructions
- Finally, include Jumps and Branches

Datapath for R-type Instructions



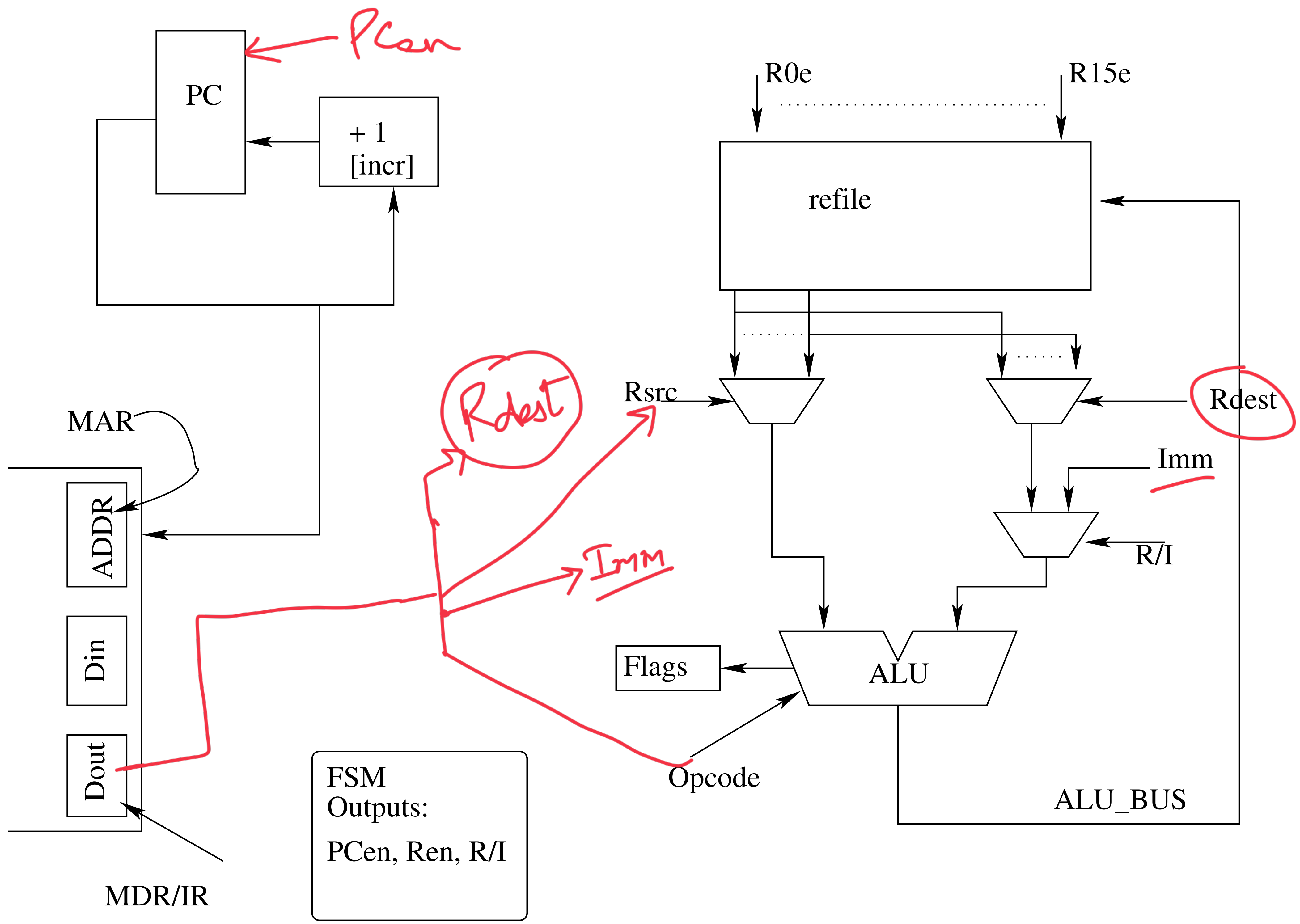
FSM
Outputs:
PCe, Ren, Rsrc,
Rdest, R/I, Opcide
Imm

FSM Design for R-type



- State S0: PCe = 0; Ren=0; Rsrc = 4'bx; Rdest=4'bx; Opcode = 8'bx; R/I = 1'bx; Imm=8'bx
    - Next state = S1
- State S1:
    - If Opcode = R-type, then NS = S2
- PCe = 0; Ren=0; Rsrc = 4'bx; Rdest=4'bx; Opcode = 8'bx; R/I = 1'bx; Imm=8'bx
- State S2: **PCe = 1**
  // Don't write PC = PC + 1 in your FSM's logic in Verilog

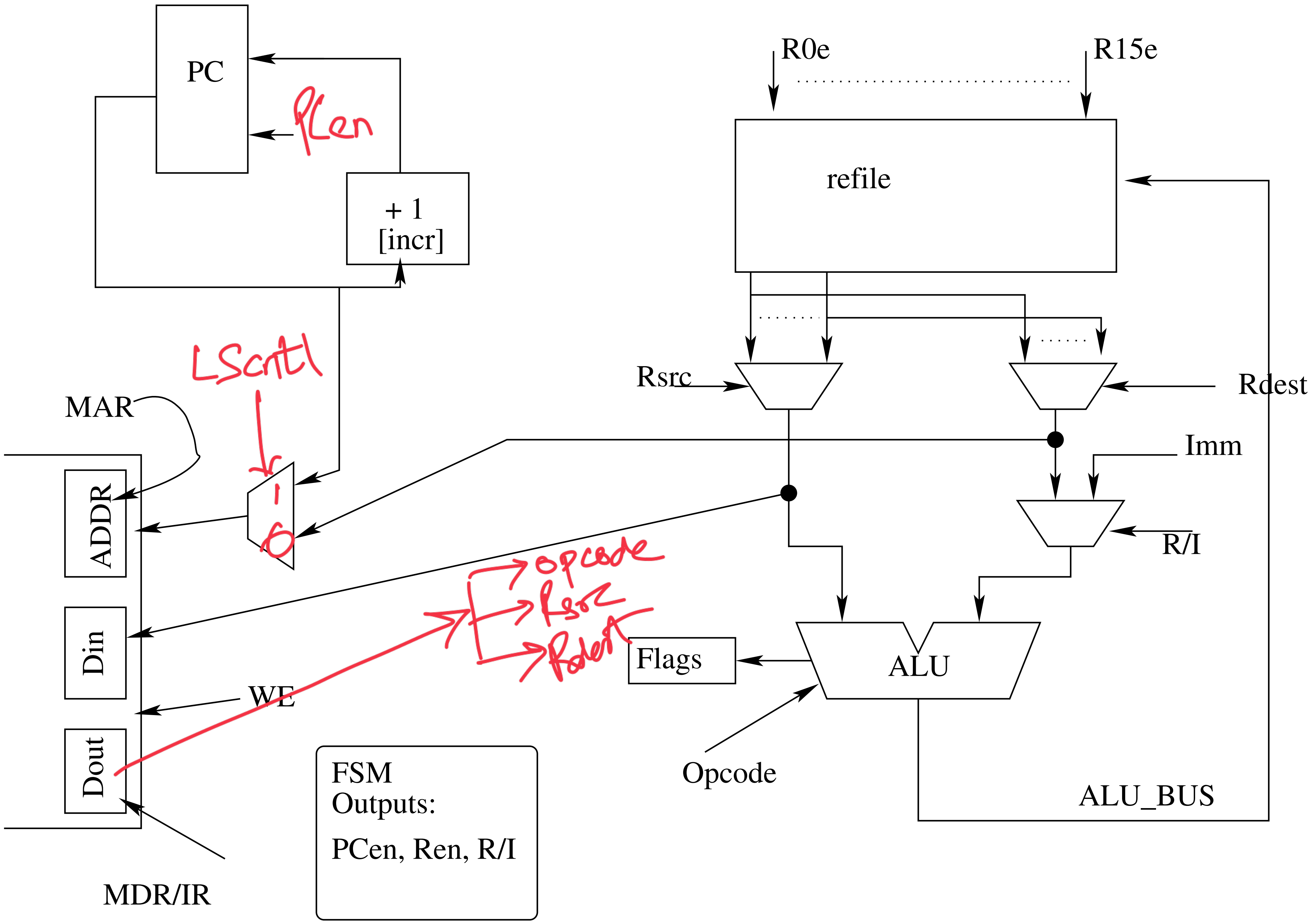
A More Structural Design



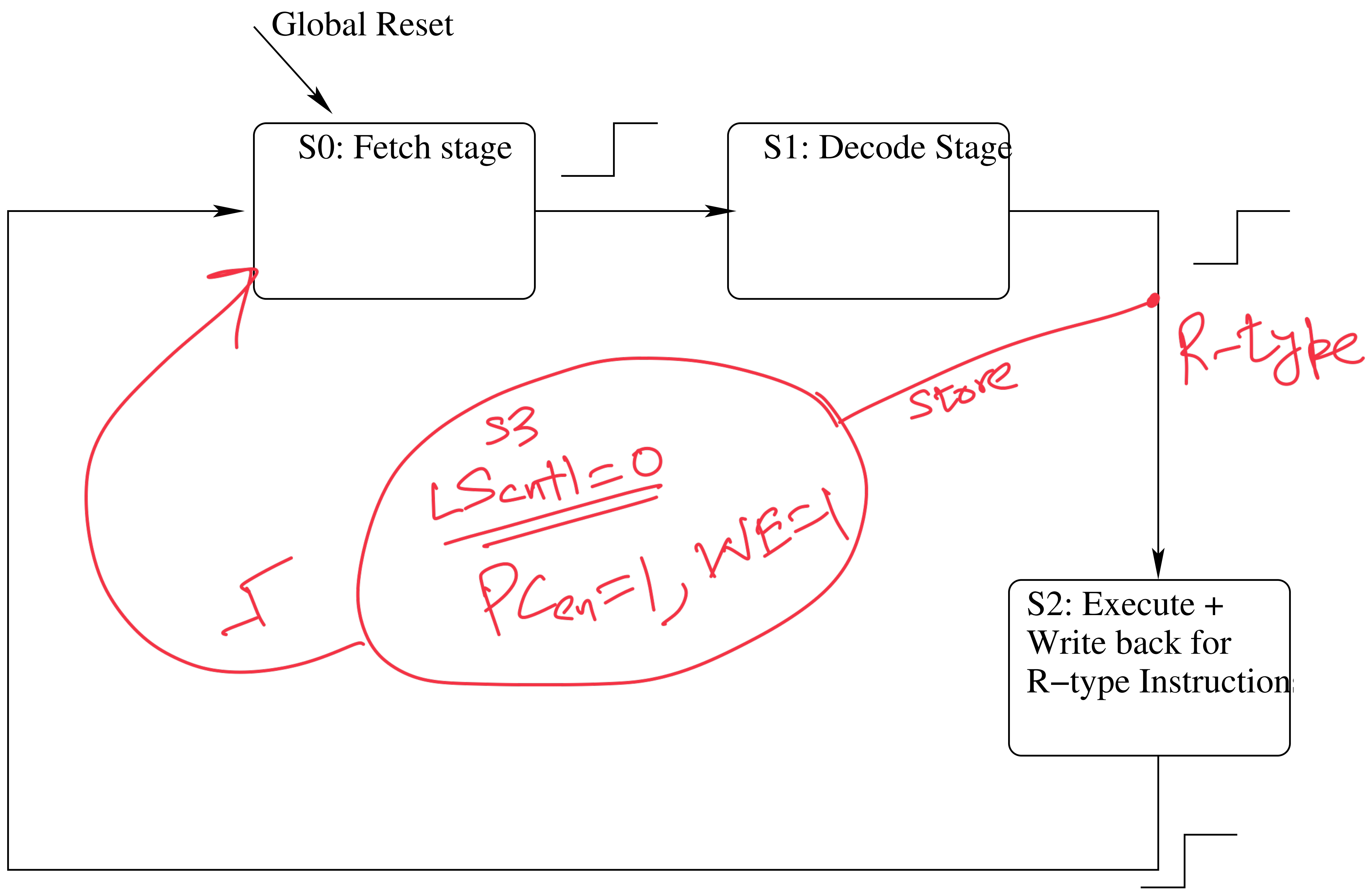
- More structural design, leaner/simpler FSM

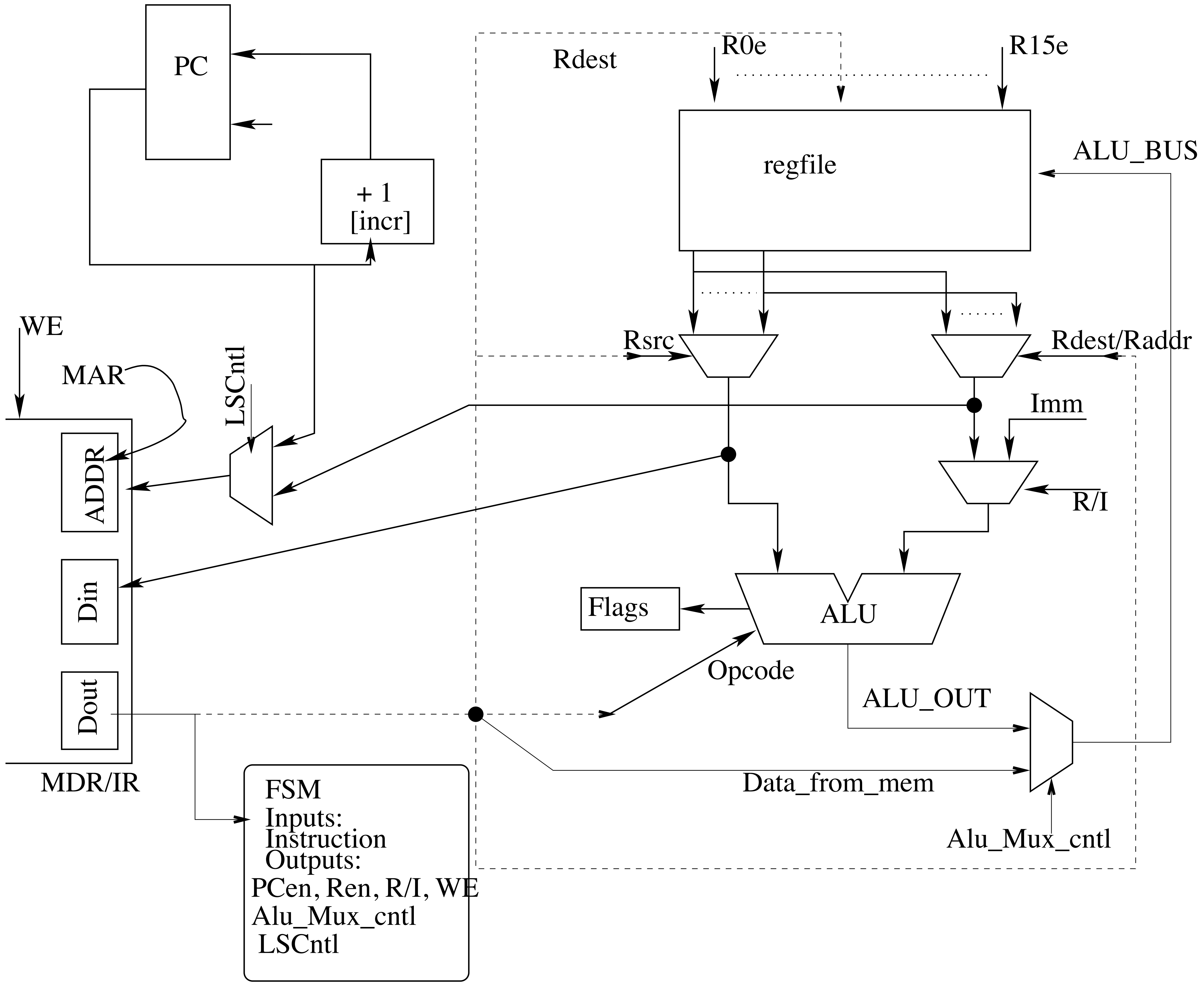
Store Instruction

- STOR Rsrc Raddr: Store data (in Rsrc) in memory at address given in Raddr



Update your FSM with Store Instruction

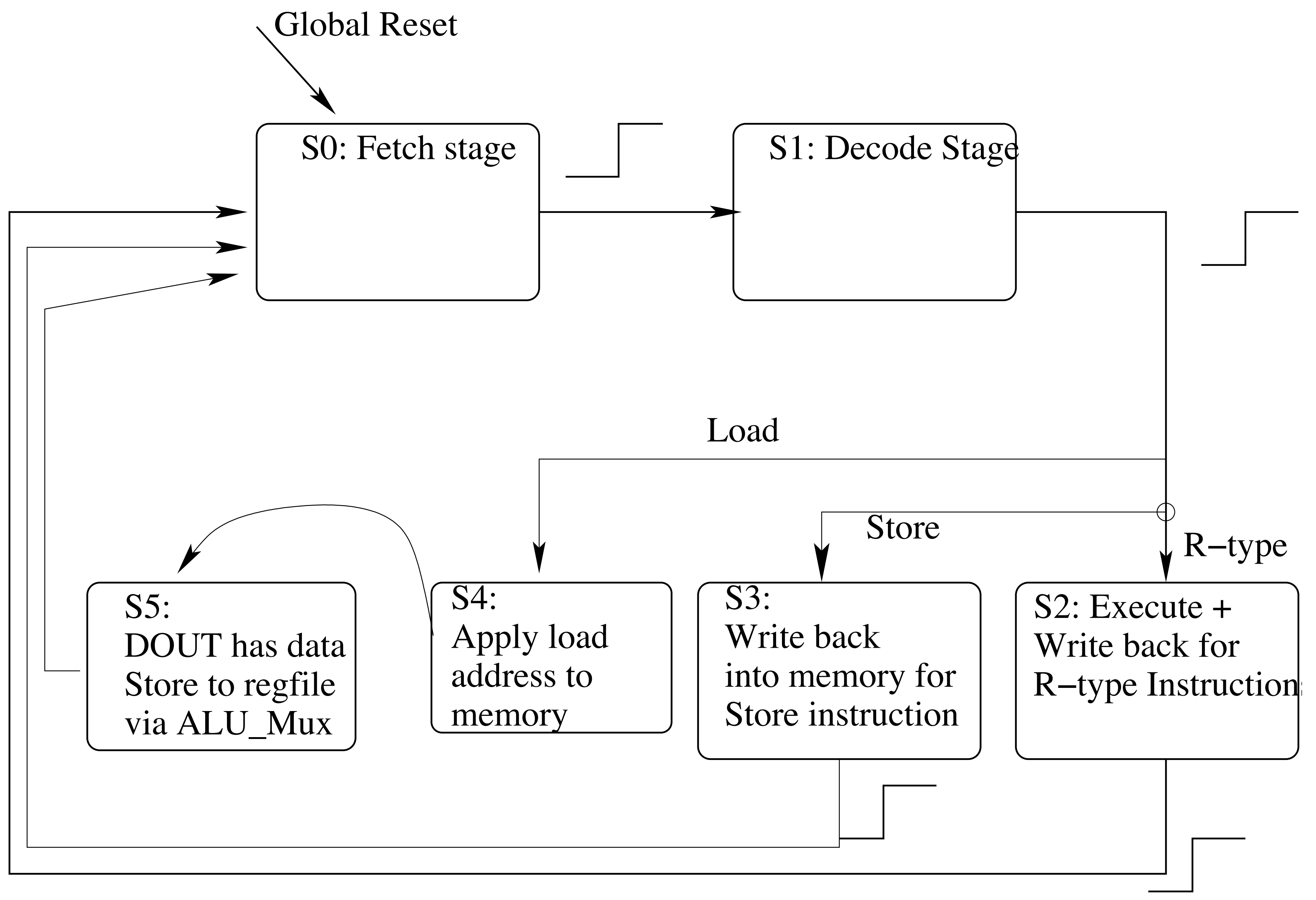



PC
+ 1
[incr]
Rdest
R0e
R15e
regfile
ALU_BUS
WE
MAR
LSCntl
Rsrc
Rdest/Raddr
Imm
R/I
ADDR
Din
Dout
MDR/IR
Flags
ALU
Opcode
ALU_OUT
Data_from_mem
Alu_Mux_cntl
FSM
Inputs:
Instruction
Outputs:
PCen, Ren, R/I, WE
Alu_Mux_cntl
LSCntl

Datapath for Load Instruction LOAD R_{dest}, R_{addr}, where R_{dest}, R_{addr} in regfile

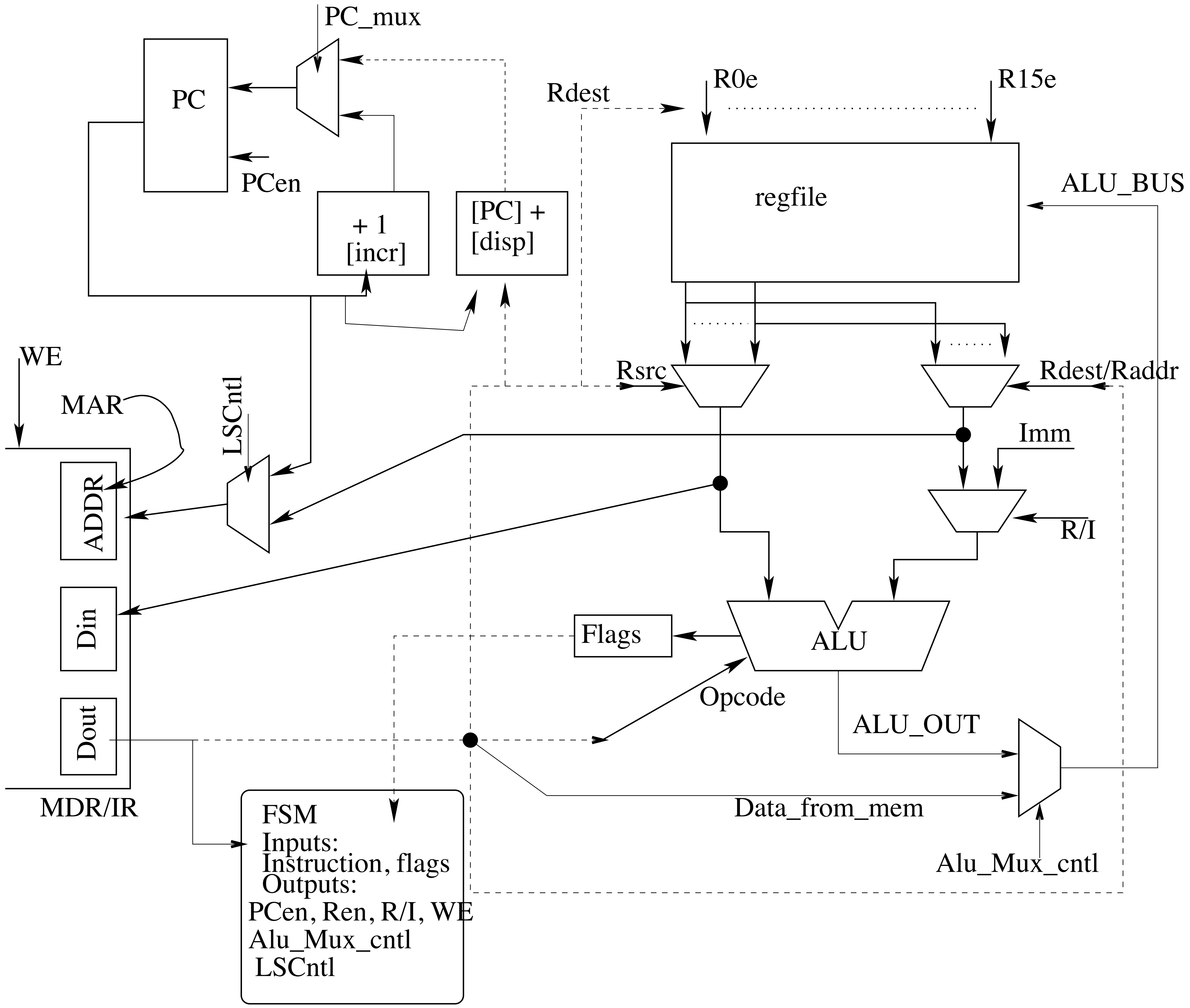
- Careful about the Opcode and Operand fields in the given instruction set architecture (ISA.pdf)
- In STORE R_{src}, R_{addr}, we have the opcode field $R_{addr} = R_{dest}$ MUX control connection
- In LOAD R_{dest}, R_{addr}, the operand fields (their roles) are sort of reversed. Here $R_{dest} = R_{en}$ (reg_enables) for writing into the regfile.
- Challenge in Load operation:
    - In states S0, S1, DOUT = Fetched instruction [LD R_{dest}, R_{addr}]
    - In state S5, Dout = Data. You may have to “register” from DOUT.
    - You can do it in the data path or in the FSM. [Your job to think about this!!]

FSM for R-type, LOAD, STORE

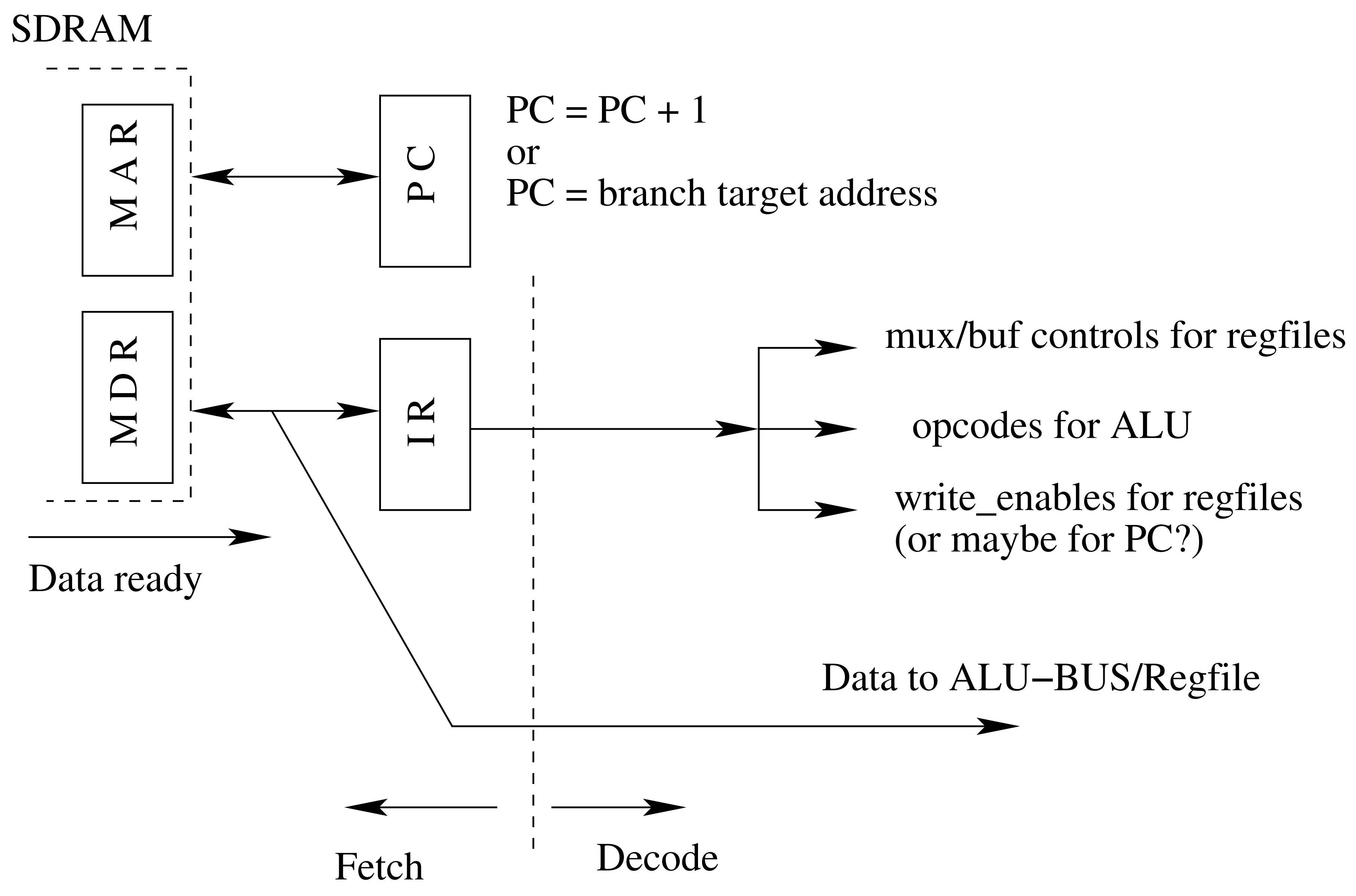


Branch and Jump Instructions

- ISA for Branch: *Bcond disp*
    - *cond* = condition codes given in ISA.pdf page 5
    - *disp*= displacement w.r.t. the PC (disp=8-bit 2’s complement)
- Lets try “*BEQZ disp*”:
    - if (Zero-flag == 1) then PC<=PC+disp
- You should do the Jump instruction yourself.

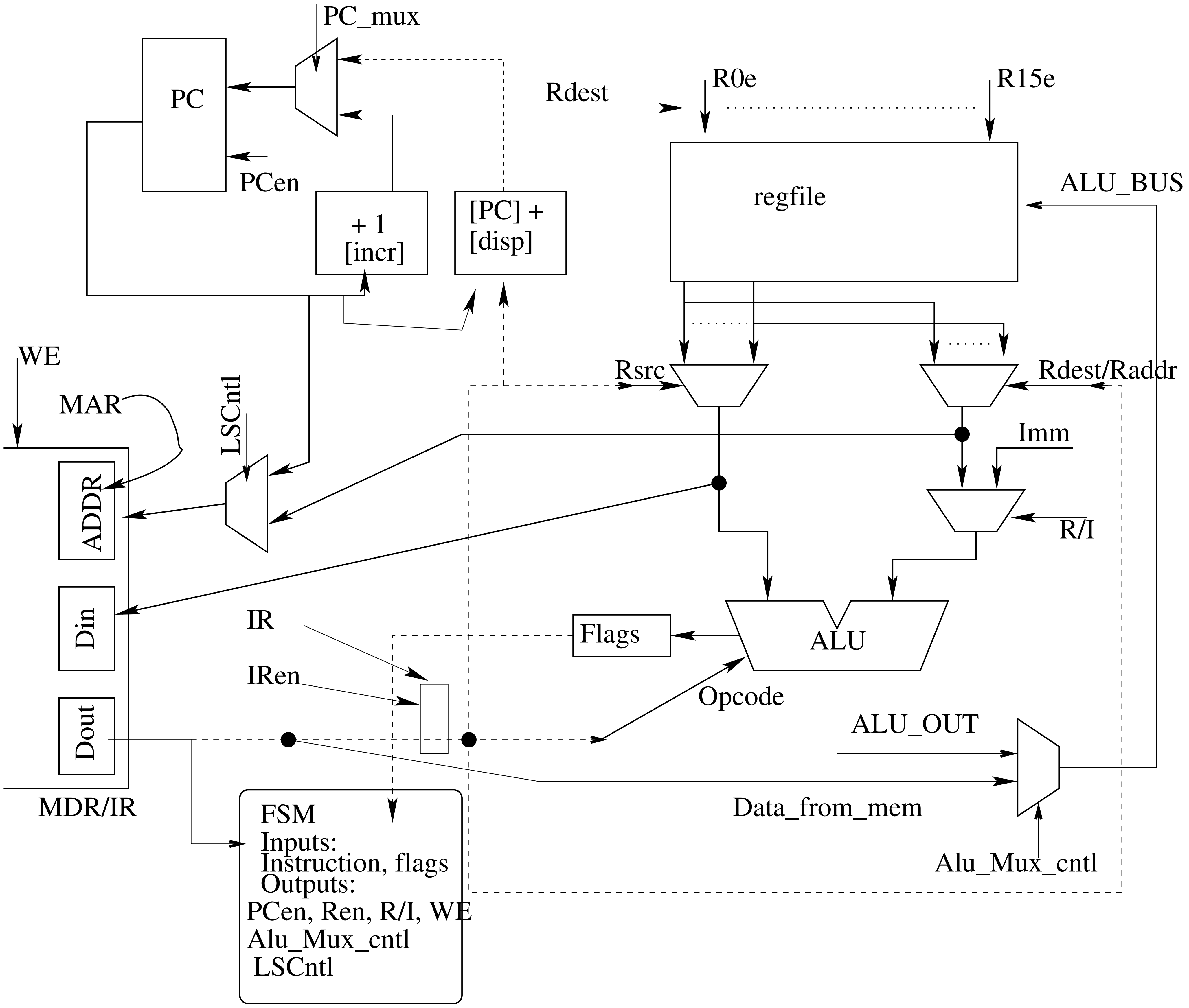

PC_mux
PC
PCen
+ 1
[incr]
[PC] +
[disp]
Rdest
R0e
R15e
regfile
ALU_BUS
WE
MAR
LSCntl
ADDR
Din
Dout
MDR/IR
Rsrc
Rdest/Raddr
Imm
R/I
Flags
ALU
Opcode
ALU_OUT
Data_from_mem
Alu_Mux_cntl
FSM
Inputs:
Instruction, flags
Outputs:
PCen, Ren, R/I, WE
Alu_Mux_cntl
LSCntl

The need for an Instruction Register (IR)



Need for an IR

- Needed mostly for a LOAD operation
- DOUT = data_out from memory
    - DOUT holds the instruction, during the instruction fetch cycle
    - DOUT holds DATA to load into regfile, during the data fetch cycle. This data overwrites the instruction, so the $R_{dest}, R_{src}, R_{addr}$ fields get overwritten, and are lost
- To avoid this, use an instruction register (IR), with an IR_enable signal. Copy instruction into IR from DOUT ($[IR] \leftarrow [D_{out}]$) after state $S1$
- In the next slide, see the final data path that accommodates: Load, Store, Reg-to-Reg, Immediate, and branch instructions
- You need to further extend this data path for the Jump instructions


PC_mux
PC
PCen
+ 1
[incr]
[PC] +
[disp]
Rdest
R0e
R15e
regfile
ALU_BUS
WE
MAR
LSCntl
ADDR
Din
Dout
MDR/IR
Rsrc
Rdest/Raddr
Imm
R/I
ALU
Flags
IR
IRen
Opcode
ALU_OUT
Data_from_mem
Alu_Mux_cntl
FSM
Inputs:
Instruction, flags
Outputs:
PCen, Ren, R/I, WE
Alu_Mux_cntl
LSCntl