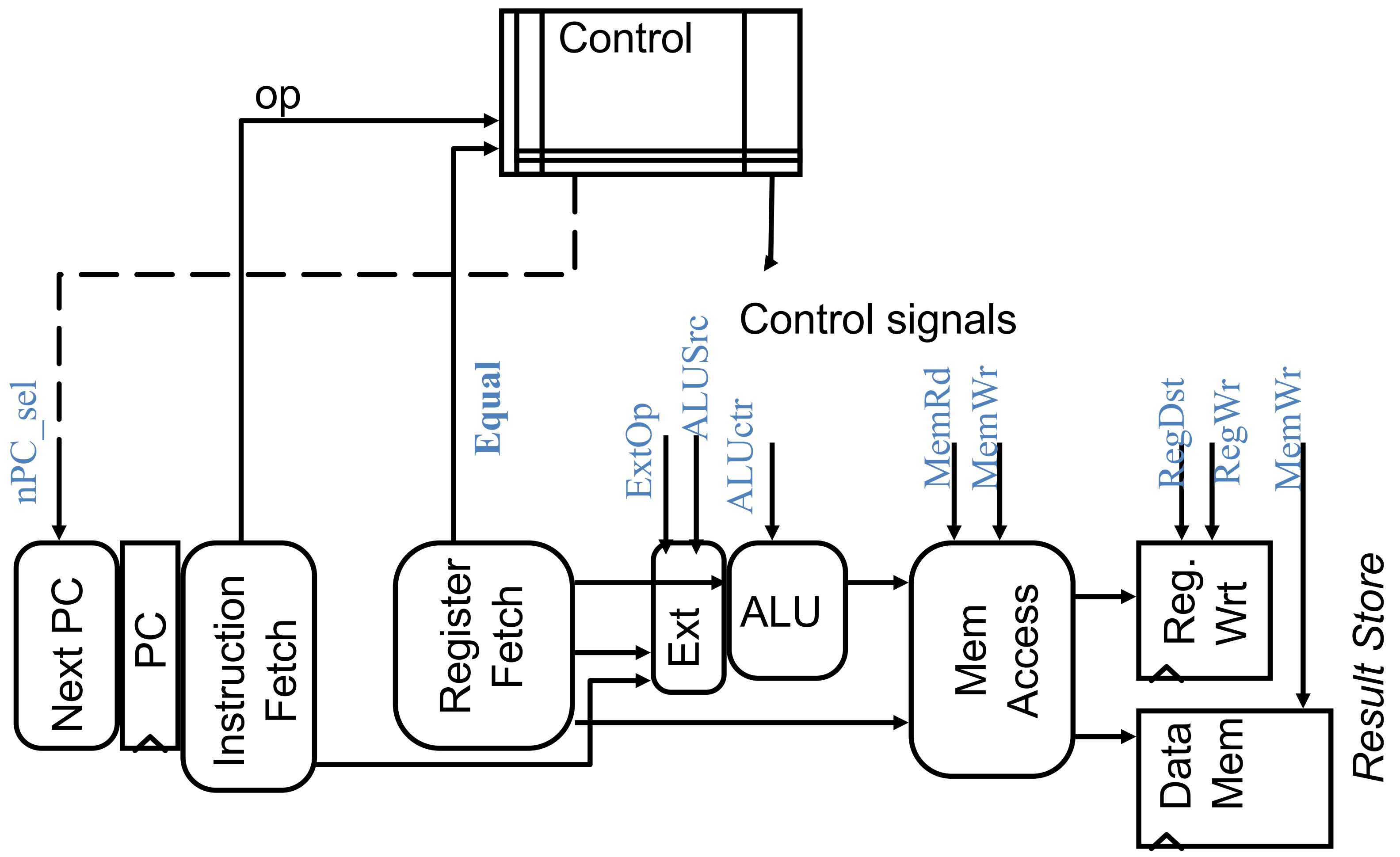
Architecture des Processeurs

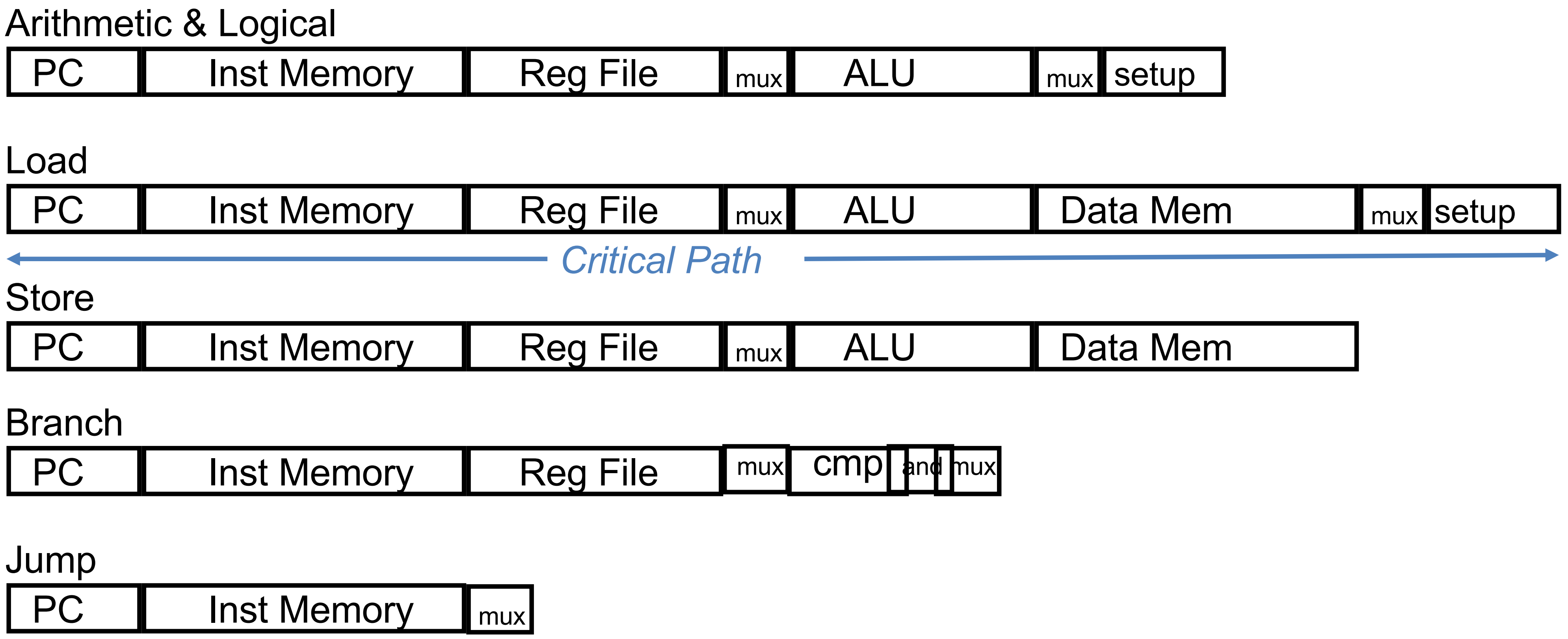
Micro-architecture d’un processeur Multi-Cycle

yann.douze@sorbonne-universite.fr

Abstract View of our single cycle processor



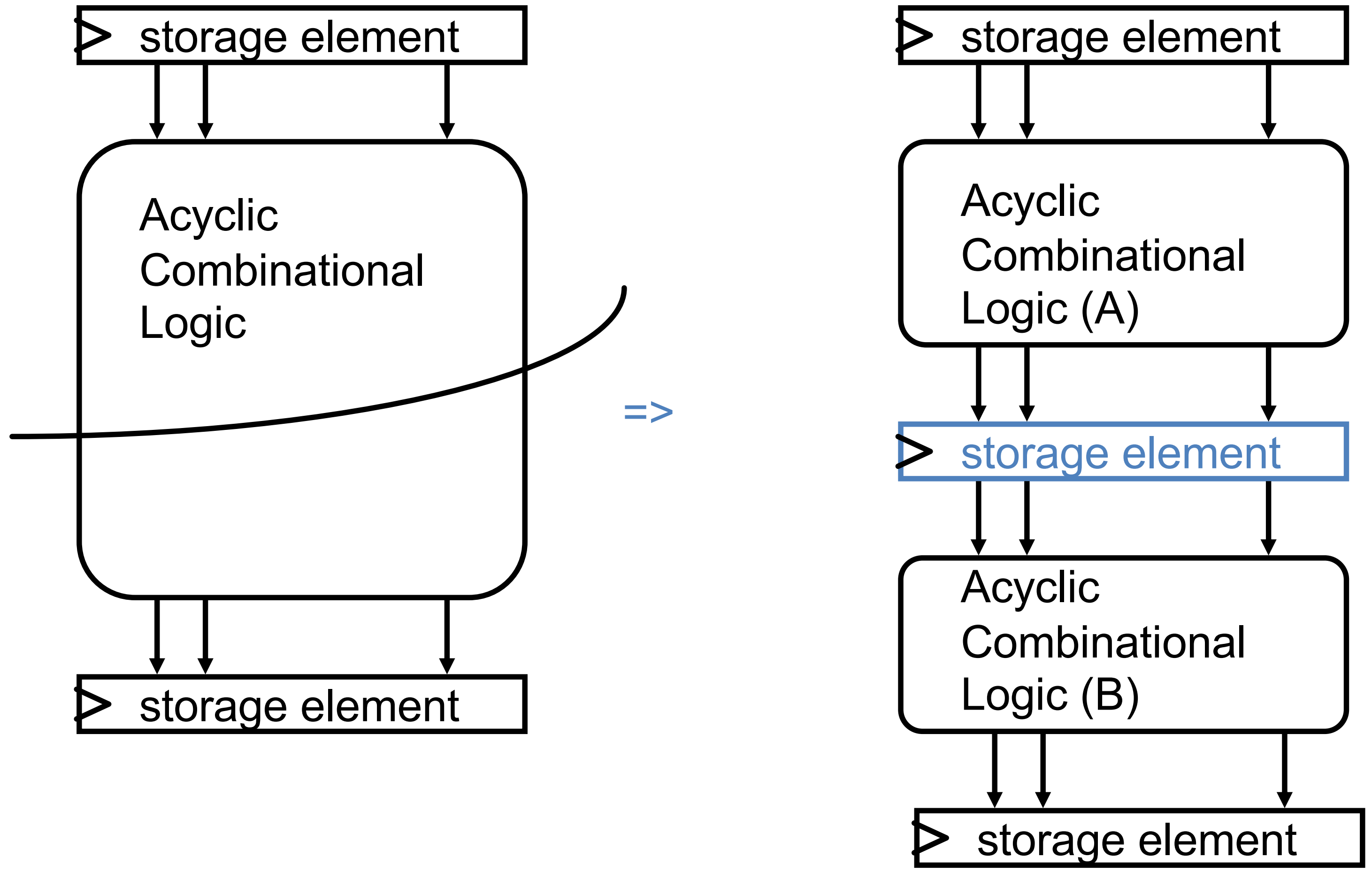
What’s wrong with our CPI=1 processor?



- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not so nice as our idealized memory
  – cannot always get the job done in one (short) cycle

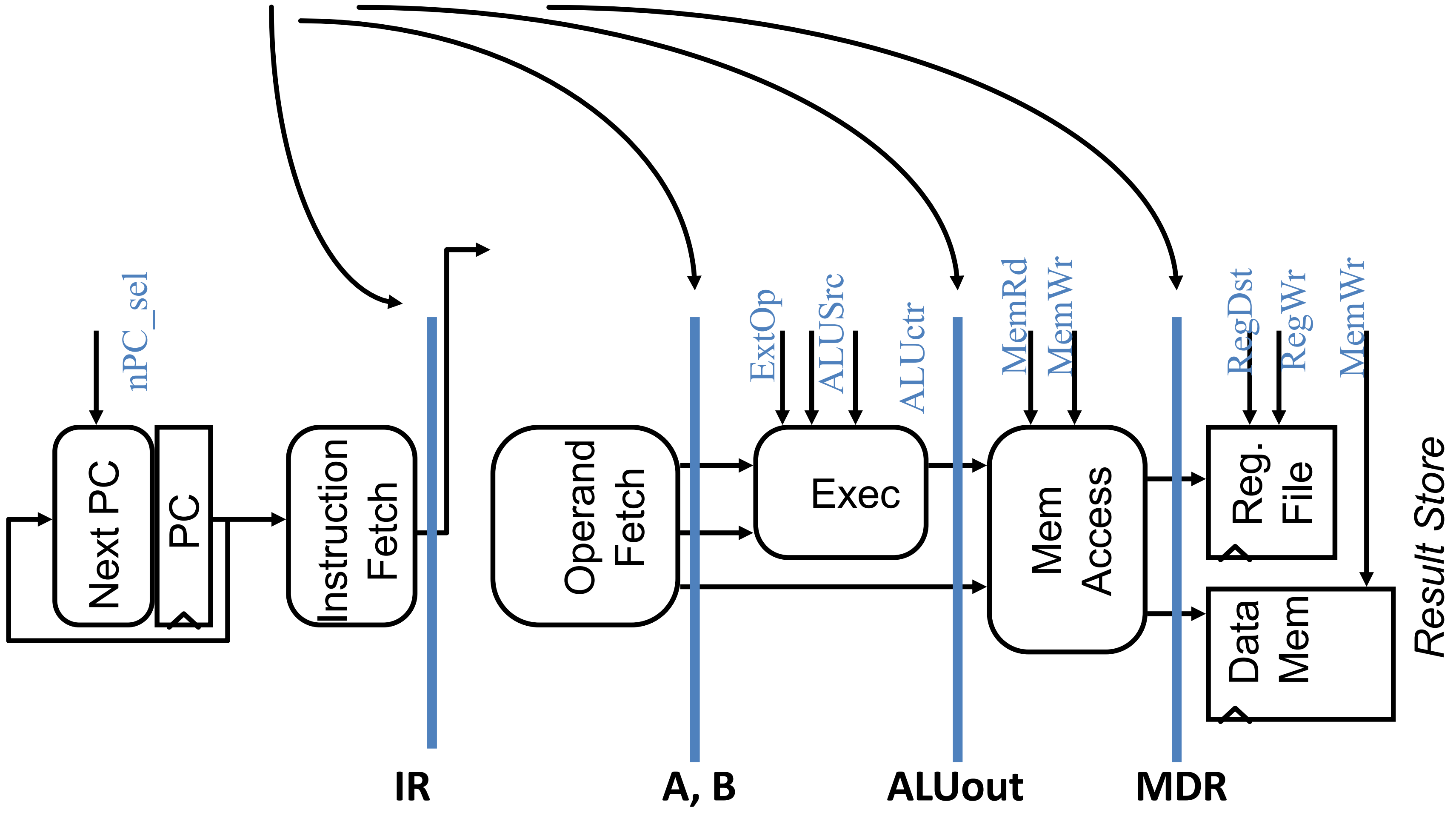
Reducing Cycle Time

- Cut combinational dependency graph and insert registers
- Do same work in two fast cycles, rather than one slow one


storage element
Acyclic Combinational Logic
storage element
=>
storage element
Acyclic Combinational Logic (A)
storage element
Acyclic Combinational Logic (B)
storage element

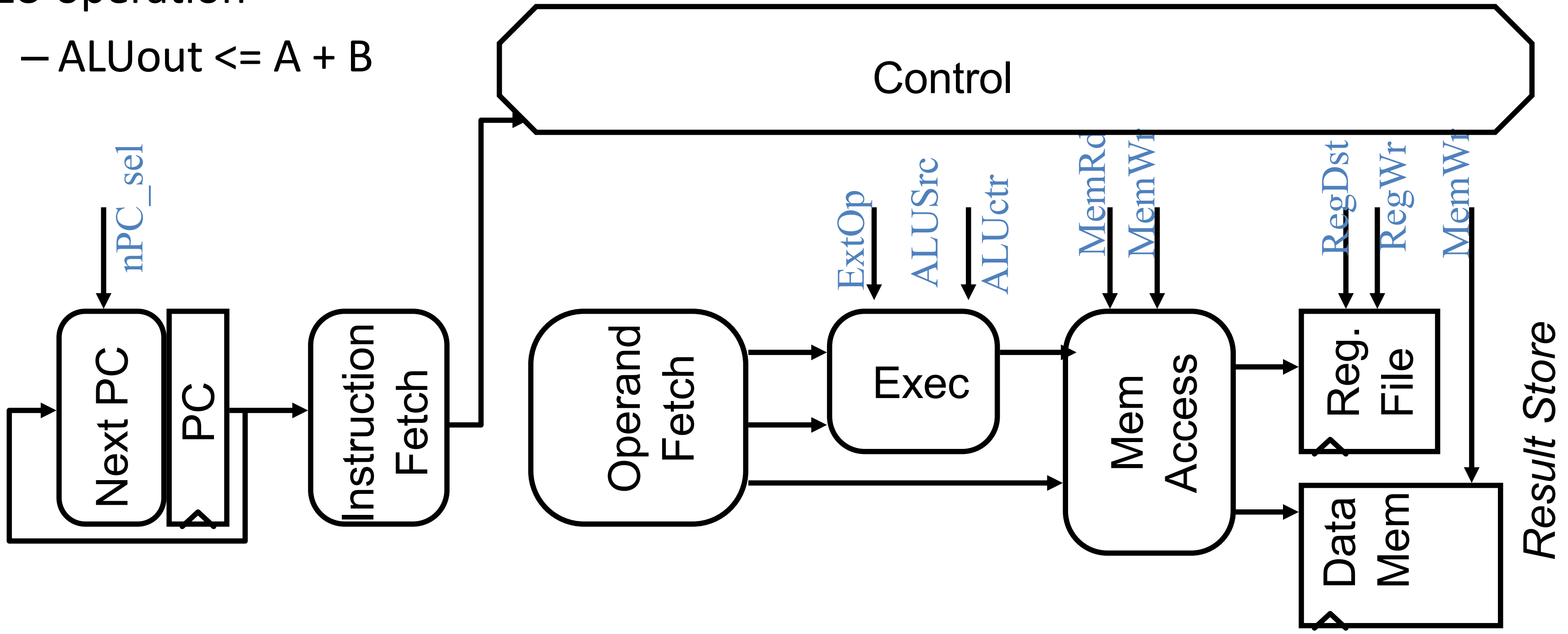
Partitioning the CPI=1 Datapath

- Add registers between smallest steps

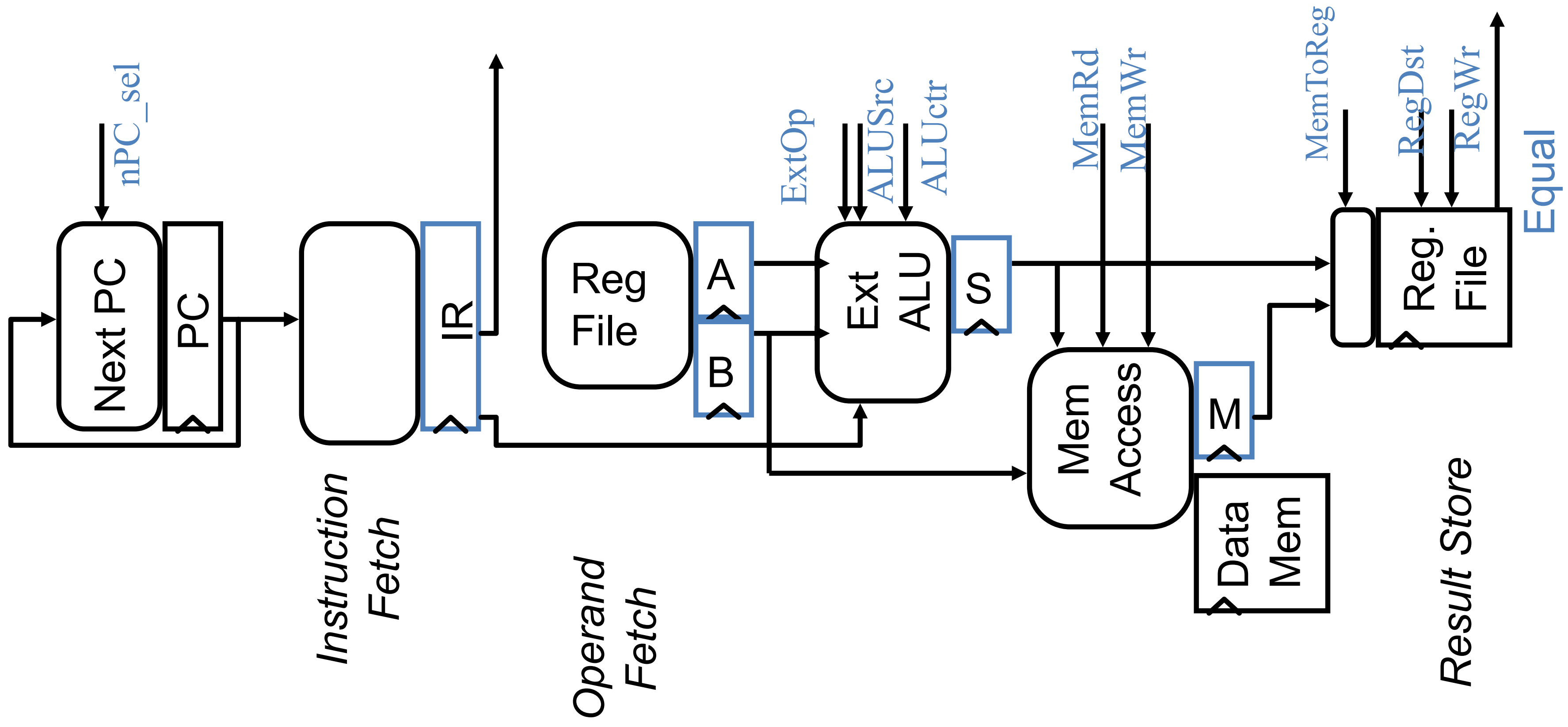


Basic Limits on Cycle Time

- Next address logic
  - PC <= branch ? PC + 1 or PC + offset
- Instruction Fetch
  - InstructionReg <= Mem[PC]
- Register Access
  - A <= R[rn], B <= R[rm]
- ALU operation
  - ALUout <= A + B



Example Multicycle Datapath



- Critical Path ?

Invoke step-by-step processor design technique

Step 1: ISA => Logical Register Transfers

Step 2: Components of the Datapath

Step 3: RTL + Components => Datapath

Step 4: Datapath + Logical RTs => Physical RTs
Logical RTs use ISA visable registers
Physical RTs use intermediate registers also

Step 5: Physical RTs => Control

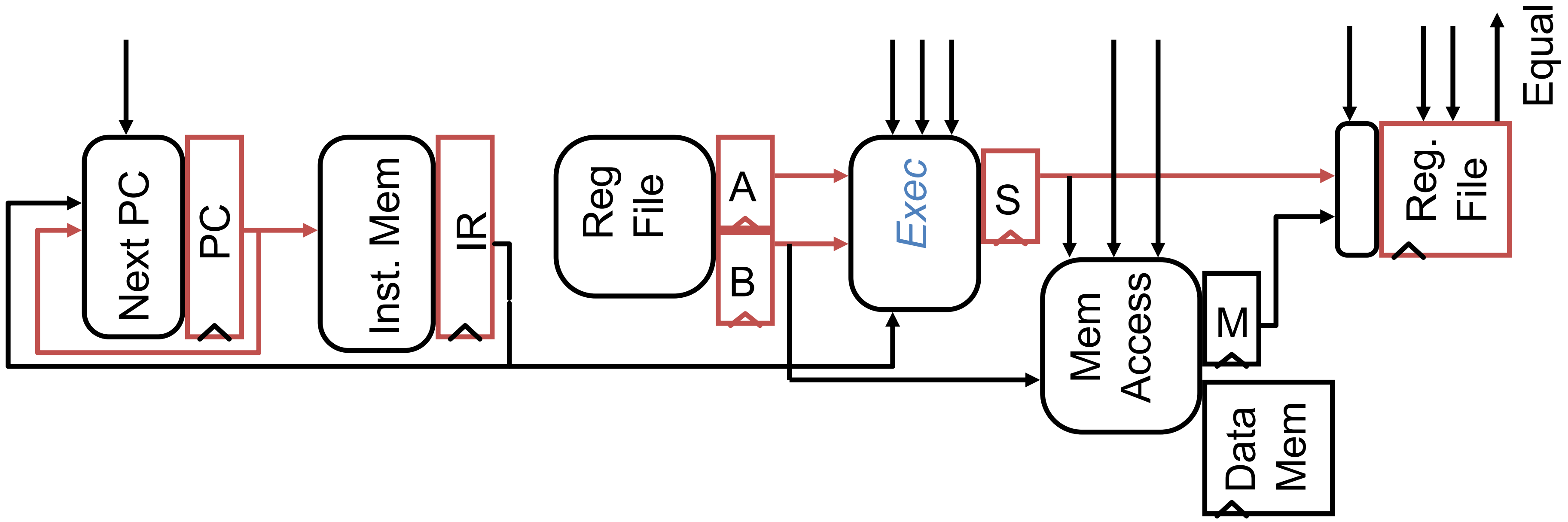
Step 4: R-rtype (add, sub, . . .)

- Logical Register Transfer

- Physical Register Transfers

inst	Logical Register Transfers
ADD	R[rd] <– R[rn] + R[rm]; PC <– PC + 1

inst	Physical Register Transfers	
	IR <– MEM[pc]	
ADD	A<– R[rn]; B <– R[rm]	
	S <– A + B	
	R[rd] <– S;	PC <– PC + 1



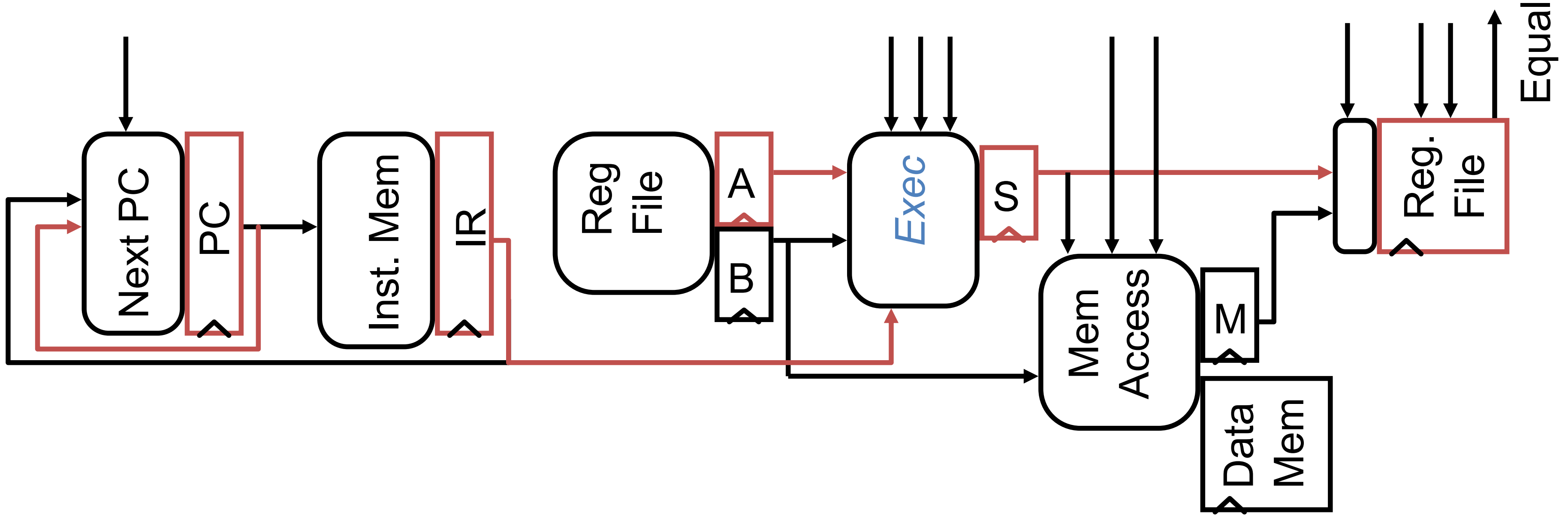
Step 4:Logical immediate

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
ORI	R[rd] <– R[rn] OR ze(imm8); PC <– PC + 1

inst	Physical Register Transfers
	IR <– MEM[pc]
ORI	A<– R[rn], B <– R[rm]
	S <– A *or* ZeroExt(imm8)
	R[rd] <– S; PC <– PC + 1

side effect

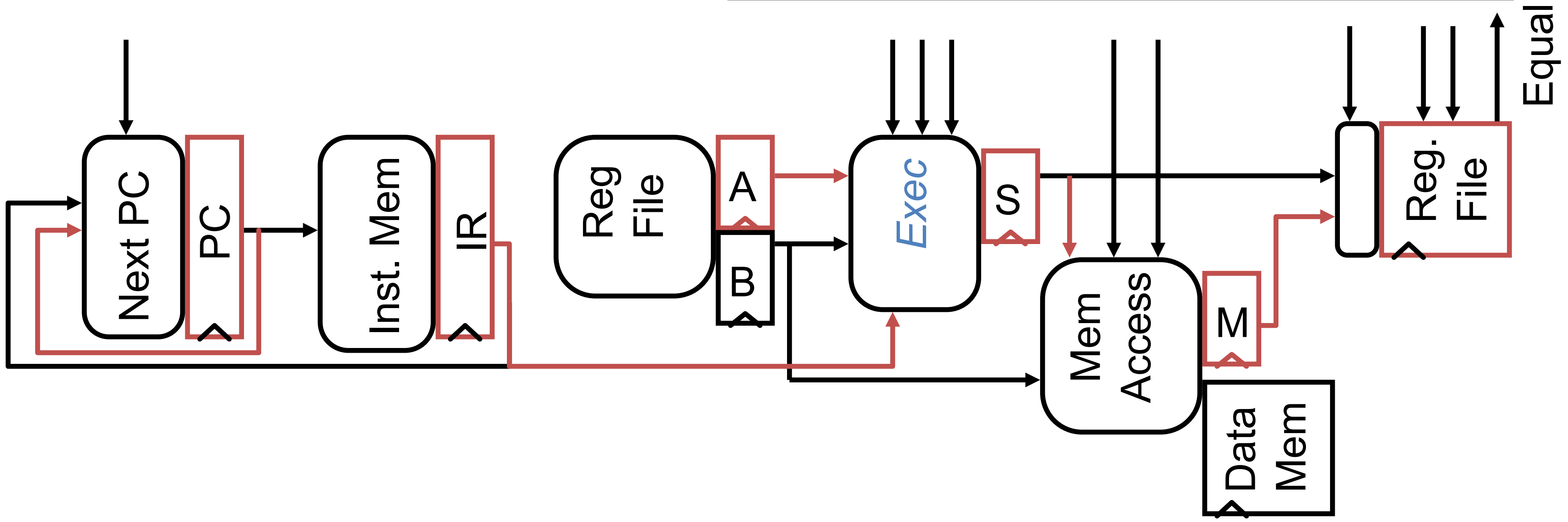


Step 4 : Load

- Logical Register Transfer
- Physical Register Transfers

inst	Logical Register Transfers
LDR	R[rd] <– MEM(R[rn] + sx(imm8);
	PC <– PC + 1

inst	Physical Register Transfers
	IR <– MEM[pc]
LDR	A<– R[rn]; B <– R[rt]
	S <– A + SignEx(imm8)
	M <– MEM[S]
	R[rd] <– M; PC <– PC + 1

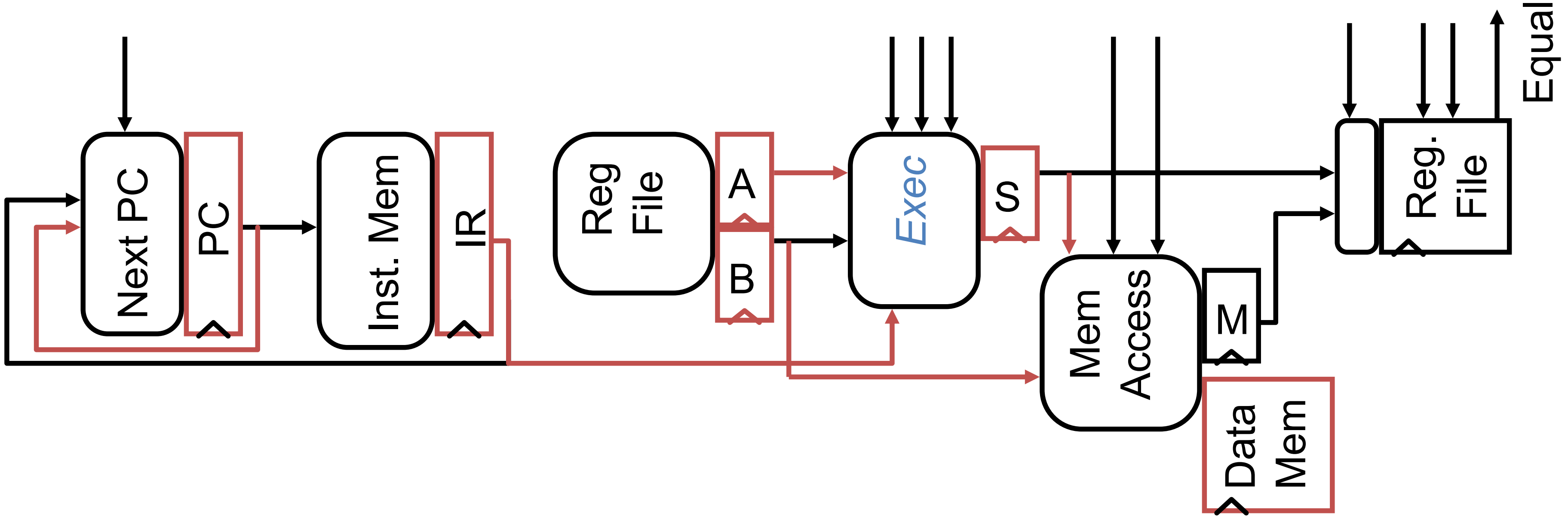


Step 4 : Store

- Logical Register Transfer
- Physical Register Transfers

inst	**Logical Register Transfers**
STR	**MEM(R[rn] + sx(imm8) <– R[rd];**
	PC <– PC + 1

inst	**Physical Register Transfers**	
	IR <– MEM[pc]	
STR	**A<– R[rn]; B <– R[rd]**	
	S <– A + SignEx(imm8);	
	MEM[S] <– B	**PC <– PC + 1**



Step 4 : Branch

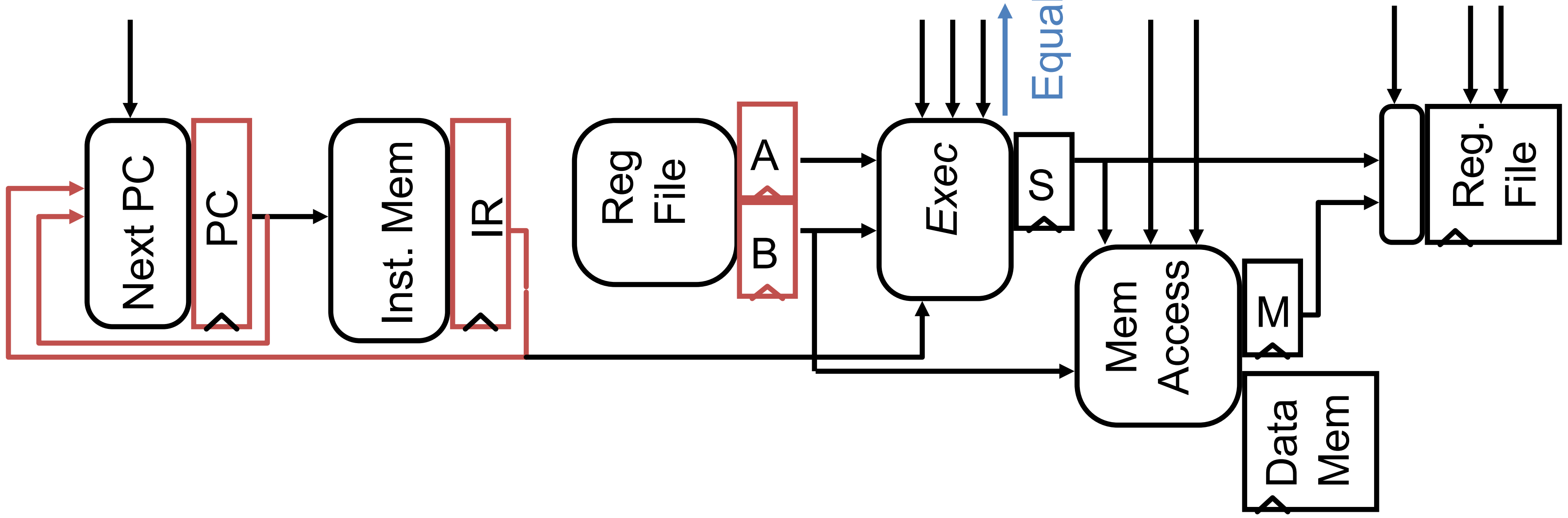
- Logical Register Transfer

inst	**Logical Register Transfers**
BEQ	**if R[rn] == R[rm]**
	then PC <= PC + sx(imm24)
	else PC <= PC + 1

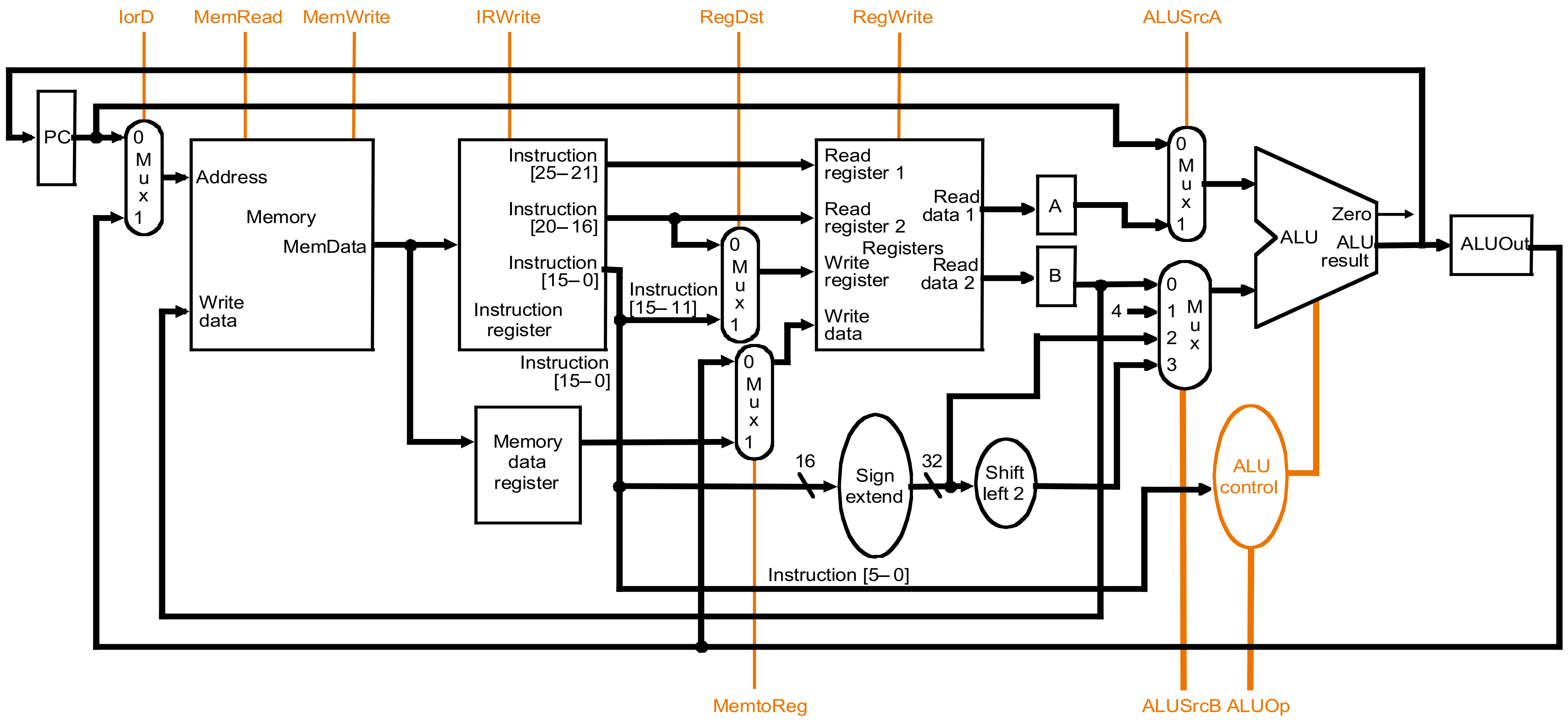
- Physical Register Transfers

inst	**Physical Register Transfers**
	IR <– MEM[pc]
	A<– R[rn]; B <– R[rm]
BEQ|$\overline{\text{Eq}}$	**PC <– PC + 1**

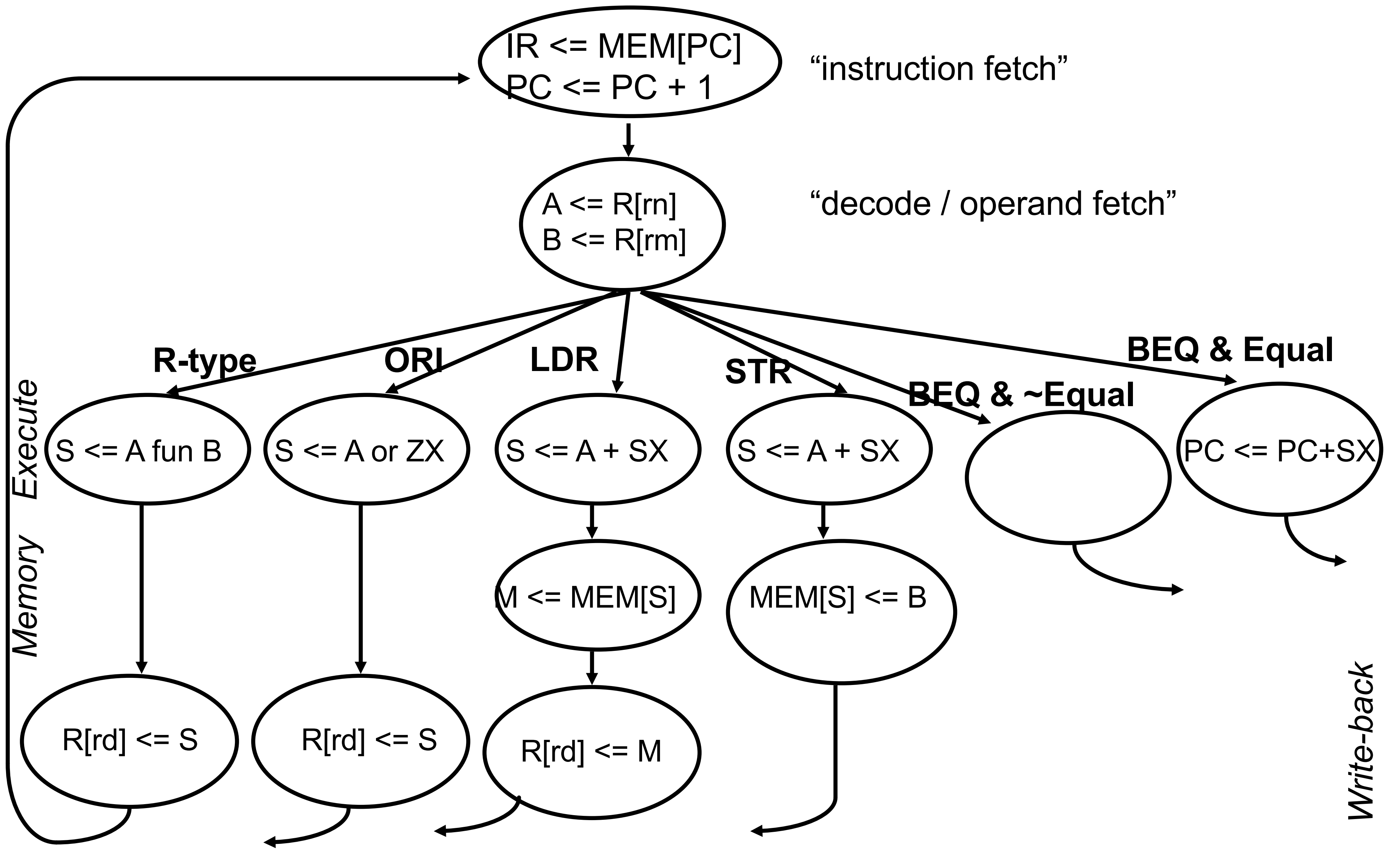
inst	**Physical Register Transfers**
	IR <– MEM[pc]
	A<– R[rn]; B <– R[rm]
BEQ|Eq	**PC <– PC + sx(imm16)**



Multiple cycle datapath with control lines



Step 4
Control Specification for multicycle datapath


IR <= MEM[PC]
PC <= PC + 1
"instruction fetch"
A <= R[rn]
B <= R[rm]
"decode / operand fetch"
R-type
ORI
LDR
STR
BEQ & ~Equal
BEQ & Equal
Execute
S <= A fun B
S <= A or ZX
S <= A + SX
S <= A + SX
PC <= PC+SX
Memory
M <= MEM[S]
MEM[S] <= B
R[rd] <= S
R[rd] <= S
R[rd] <= M
Write-back