Multiple Cycle Processor

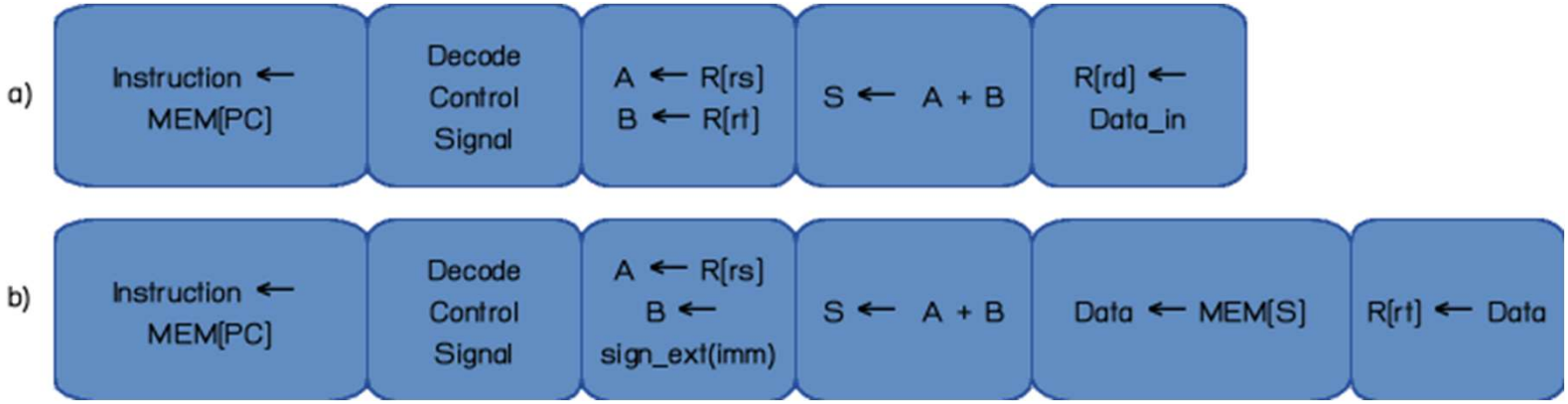
Chapter 5

Multiple Cycle Processor

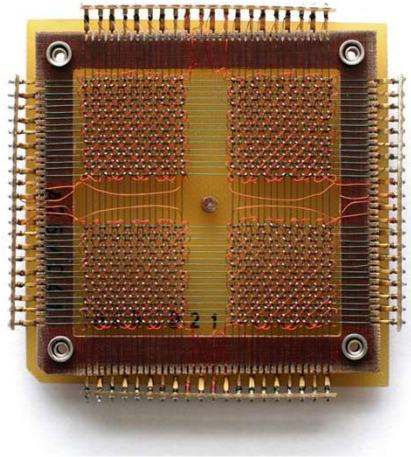
- ★ What is wrong with Single Cycle?
- ★ Memory Access Time
- ★ How to reduce cycle time
- ★ Data Path Partitioning
- ★ Logical RTL vs. Physical RTL
- ★ Control Model
- ★ Implementing the Control
- ★ Performance Evaluation

What is wrong with Single Cycle?

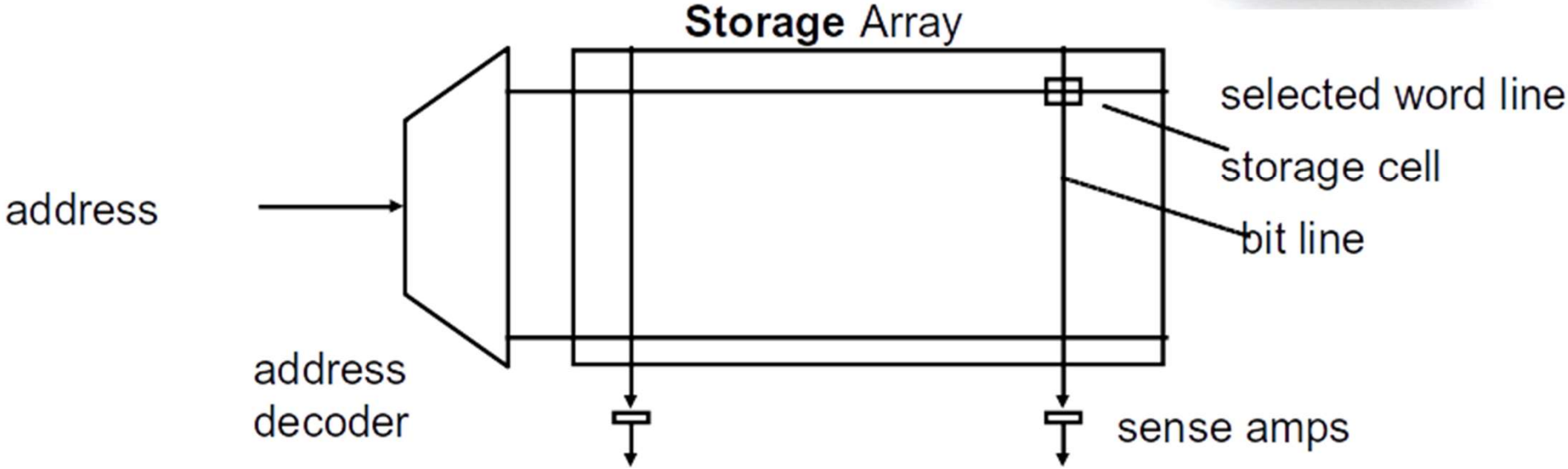
- ★ Long Cycle Time
- ★ All instructions take as much time as the slowest.
- ★ Single Cycle forces all instructions to wait for the critical path.
- ★ Real memory is not so nice as our idealized memory (take several cycles).



Memory Access Time

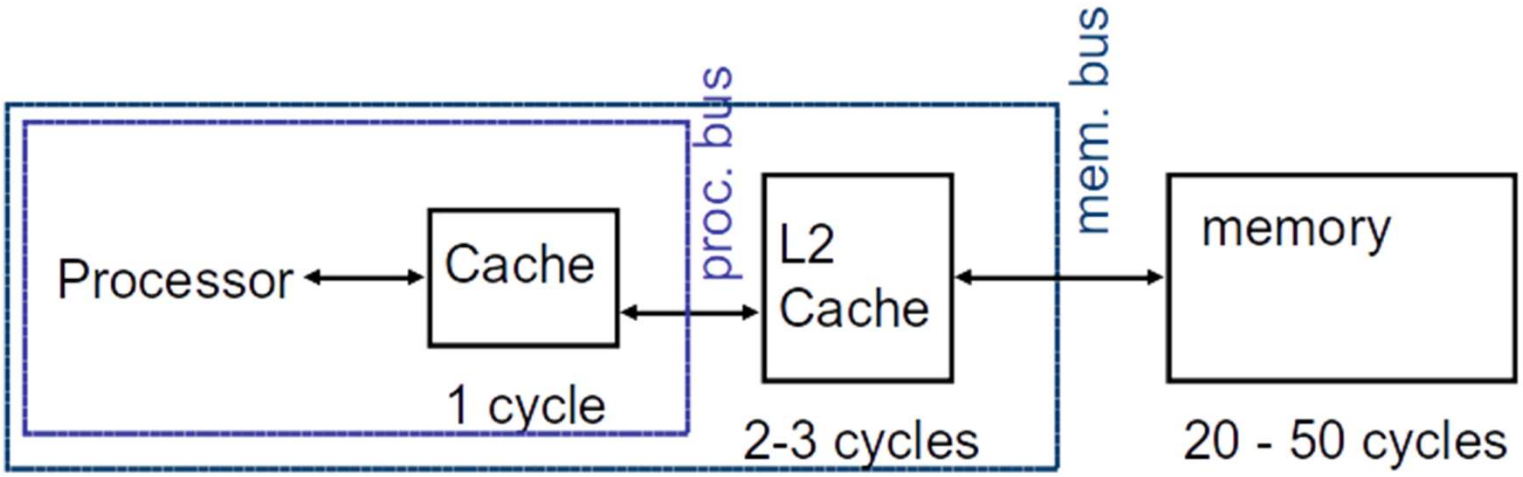


- Physical rules of memory:
  - Fast memories are small.
  - Large memories are slow.



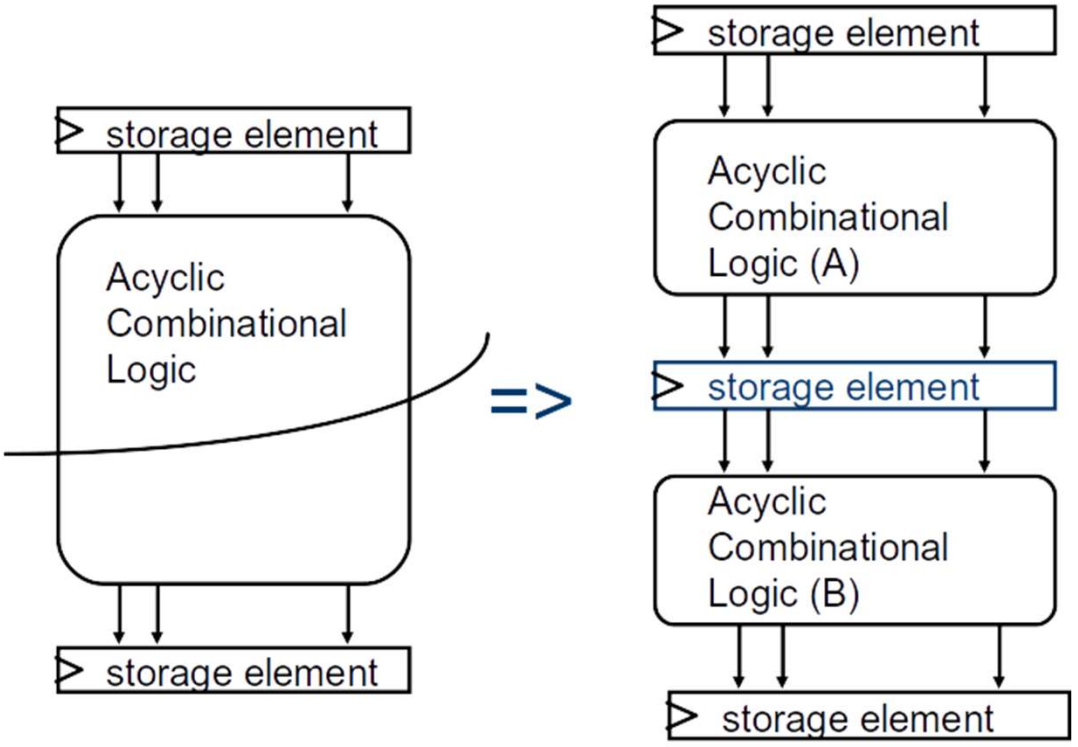
Memory Access Time (ctd.)

- ★ With memory hierarchy (CACHE), we may assume that read/write can be done in one cycle (for now).
- ★ We will revisit this concept later in chapter 8.



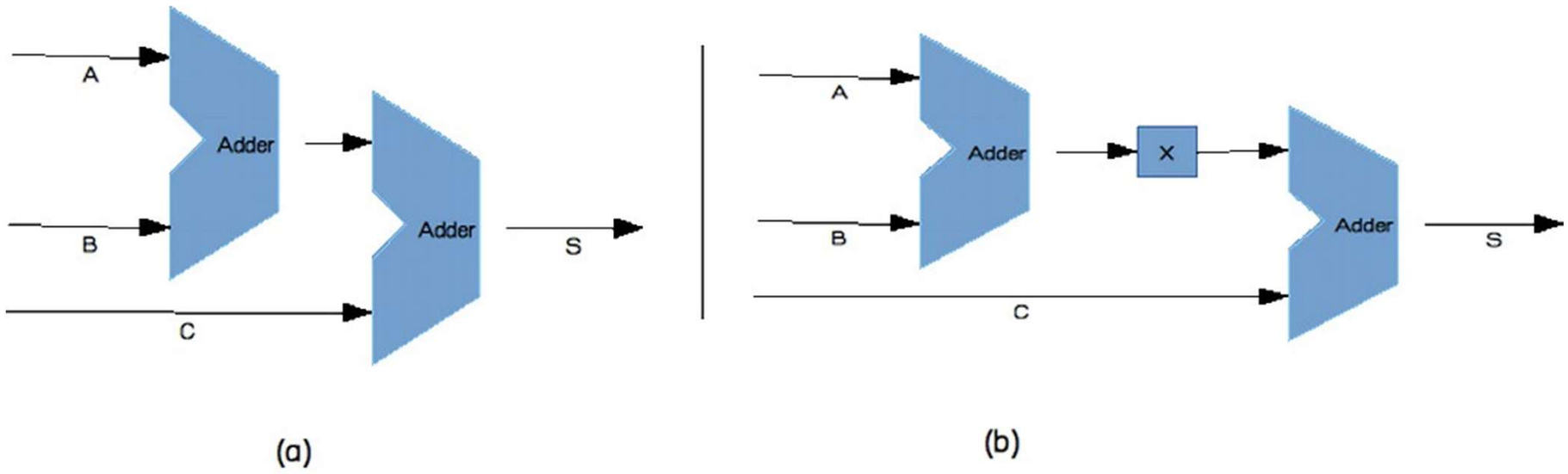
How to reduce cycle time?

- ★ Partition combinational dependency graph and insert intermediate register (latch).
- ★ Use several (fast) cycles, rather than one slow cycle.
- ★ Some cycles can be skip for certain instructions.



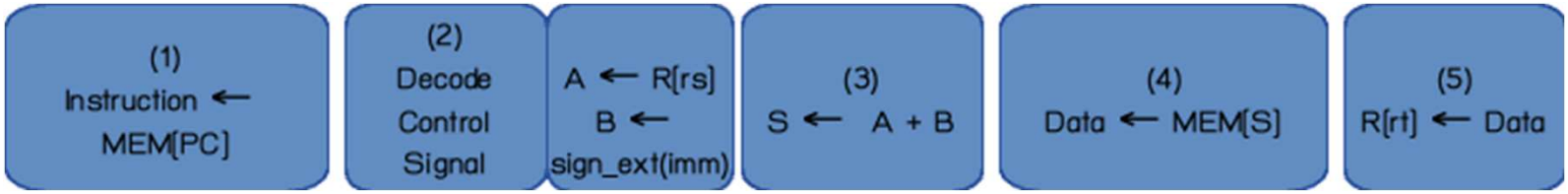
Datapath Partitioning

★ Add intermediate register(s) in between.



Basic Limits on Cycle Time

- ★ Each part should take similar amount of times
- ★ For our design, 5 parts
  - Fetch (Instruction←MEM[PC]; PC←PC+4)
  - Decode (A← ; B←)
  - Execution (S← A+B)
  - Memory Access (Data ← MEM[S])
  - Write back (R[rt]←Data)

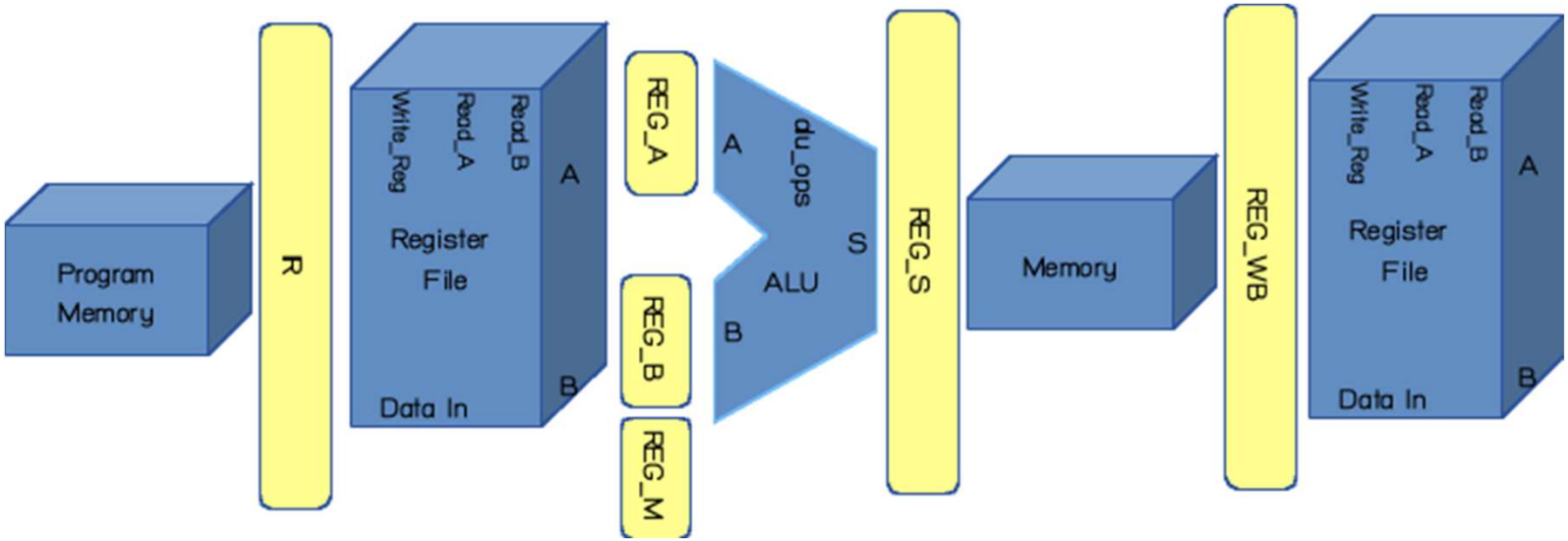


Potential Speedup

- ★ Assume that cycle time of the original single cycle processor is 210ns. If multiple cycle can reduce the cycle time to 45ns, calculate the new average CPI that will give us at least the same performance as the original one.

New Datapath

★ Add registers between smallest steps



Our Design Steps

1. ISA Analysis
2. Components design
3. Assembly Datapath
4. Datapath + Logical RTs → Physical RTs
5. Physical RTs → Control

ORI

- Logical RTL:
  - R[rt] ← R[rs] | zero_ext(imm); PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	REG_A ← R[rs] REG_B ← zero_ext(imm)
3	REG_WR ← REG_S ← REG_A | REG_B
4	R[rt] ← REG_WR

ORUI

- Logical RTL:
  - R[rt] ← R[rs] | zero_pad(imm); PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	REG_A ← R[rs] REG_B ← zero_pad(imm)
3	REG_WR ← REG_S ← REG_A | REG_B
4	R[rt] ← REG_WR

Add

- Logical RTL:
  - R[rd] ← R[rs] + R[rt]; PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	REG_A ← R[rs] REG_B ← R[rt]
3	REG_WR ← REG_S ← REG_A + REG_B
4	R[rt] ← REG_WR

LW

- Logical RTL:
  - R[rt] ← MEM[R[rs] + sign_ext(imm)]; PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	REG_A ← R[rs] REG_B ← sign_ext(imm)
3	REG_S ← REG_A + REG_B
4	REG_WR ← MEM[REG_S]
5	R[rd] ← REG_WR

SW

- ★ Logical RTL:
  - MEM[R[rs] + sign_ext(imm)] ← R[rt]; PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	REG_A ← R[rs] REG_B ← sign_ext(imm) REG_M ← R[rt]
3	REG_S ← REG_A + REG_B
4	MEM[REG_S] ← REG_M

BEQ

- Logical RTL:
  - If (R[rs] == R[rt]) then
  - PC ← PC + 4 + (sign_ext(imm) * 4)
  - else PC ← PC + 4

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	(decode)

3	If (R[rs] == R[rt]) THEN
	PC ← PC + (sign_ext(imm) * 4
	)
	ENDIF

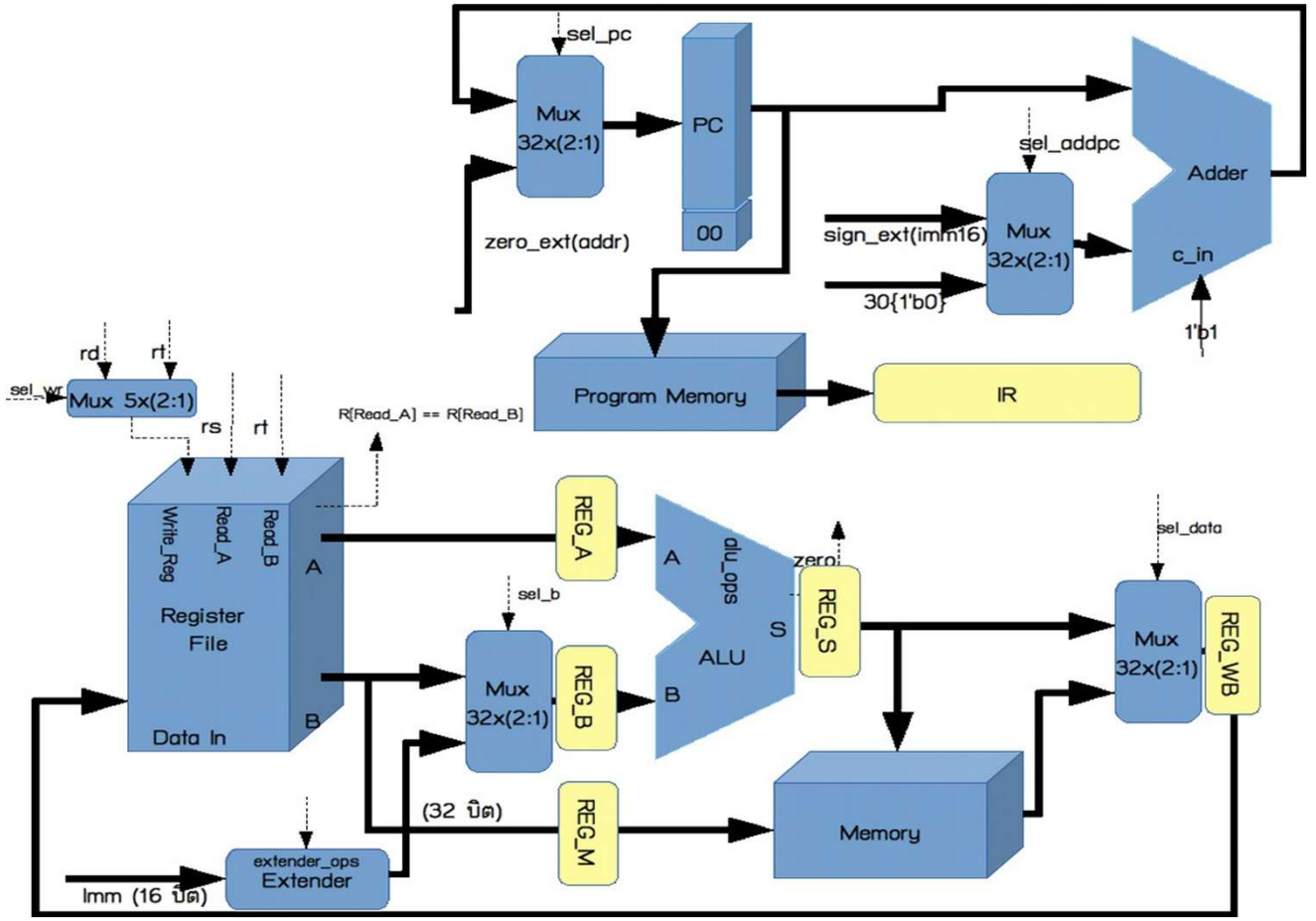
JMP

- Logical RTL:
  - PC ← (addr * 4)

Cycle	Physical RTL
1	IR ← MEM[PC] PC ← PC + 4
2	(decode)
3	PC ← addr * 4

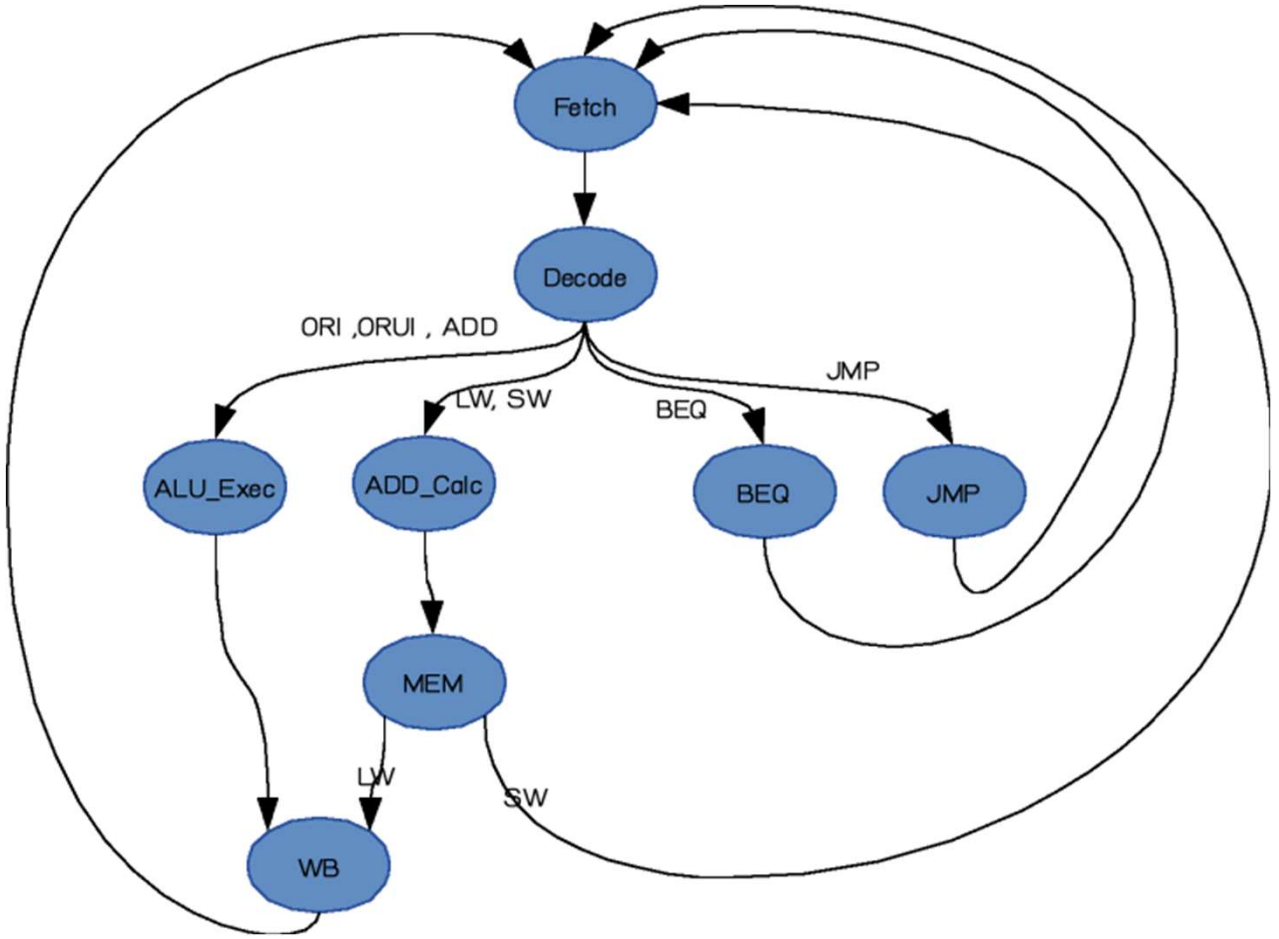
Complete Data Path

- ★ We may:
  - ALU for adding PC
  - Combine Memory



Control Path

- ★ State Diagram



Control Signal

State	Signal	Next State
Fetch	IR ← MEM[PC]	Decode

Decode	IF (op[5..4] == 01) THEN ; I-Type
	sel_wr ← 1
	sel_b ← 1
	ENDIF
	(wait for decoder)
		ALU_Exec
		ADD_Calc
		BEQ
		JMP

ALU_Exec	IF (opcode in {ORI, ORUI}) THEN
	alu_ops ← OR
	ELSE
	alu_ops ← ADD
	ENDIF	

ADD_Calc	alu_ops ← ADD	MEM

MEM	sel_data ← 1
	IF (op[2] == 1) THEN
	mem_wr ← 1
	ENDIF
		WB

WB	reg_wr=1	Fetch

BEQ	If (R[Read_A] == R[Read_B]) THEN
	sel_addpc ← 1
	END IF
		Fetch

JMP	sel_pc ← 1	Fetch

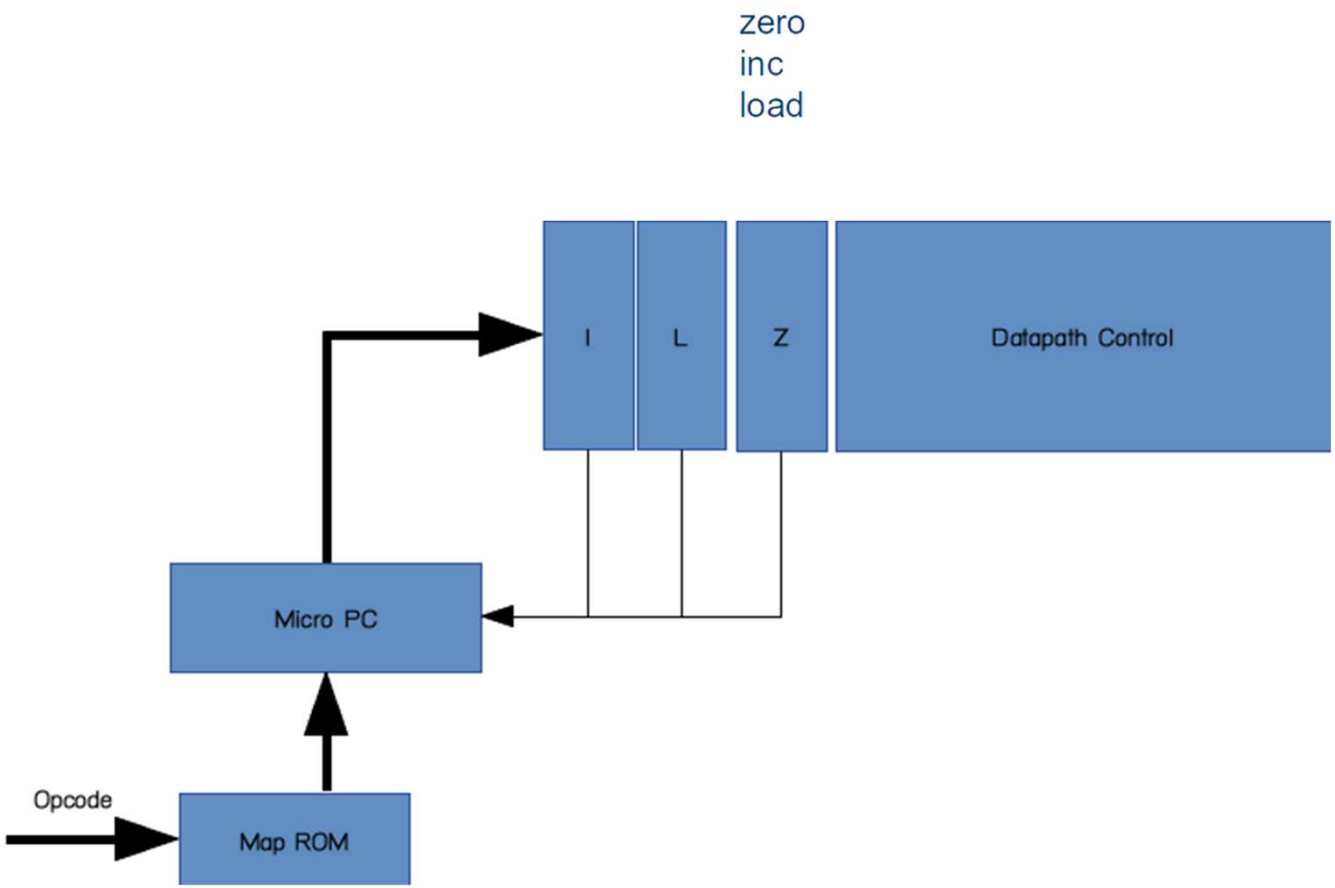
Control Implementations

Sequential Circuits

- ★ State assignment
- ★ Truth Table (Next-state function)

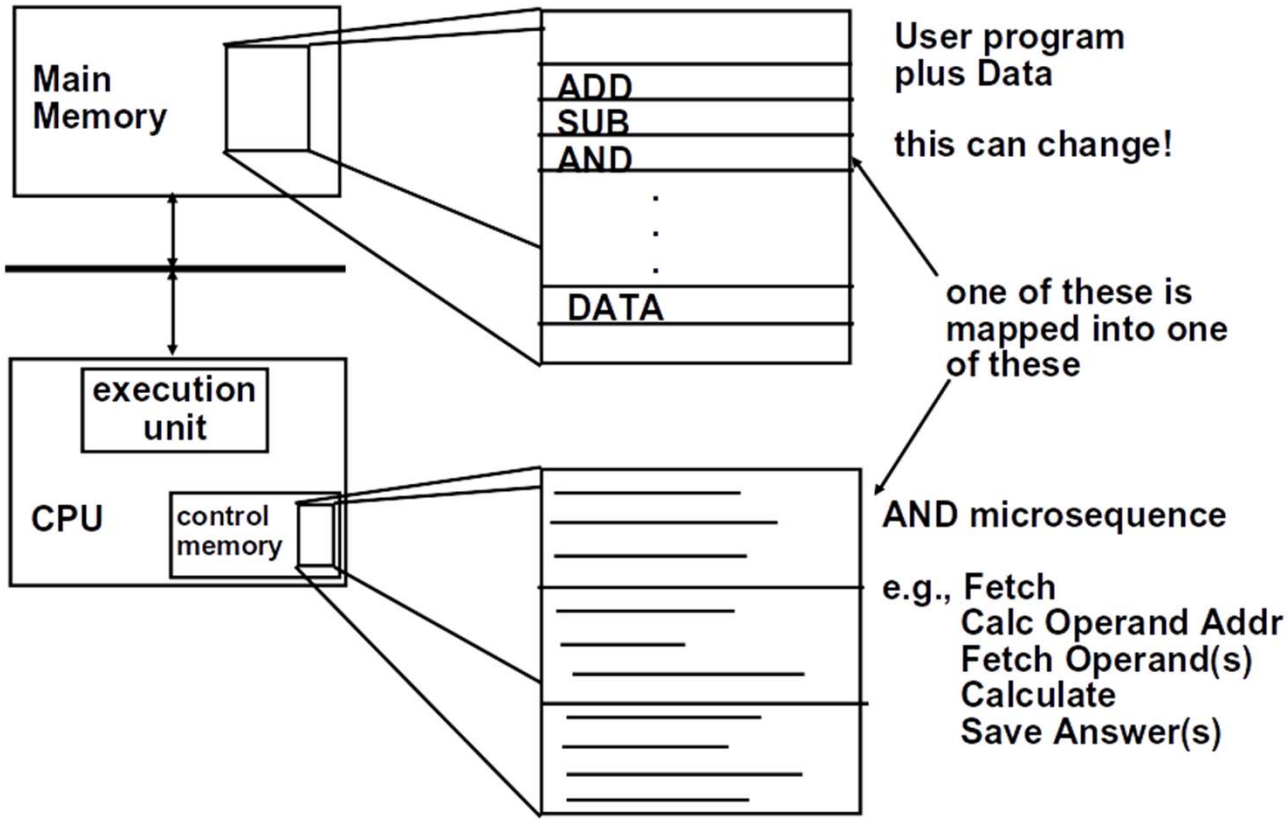
Microprogram

- ★ Microsequencer
- ★ Each microinstruction specify control signals



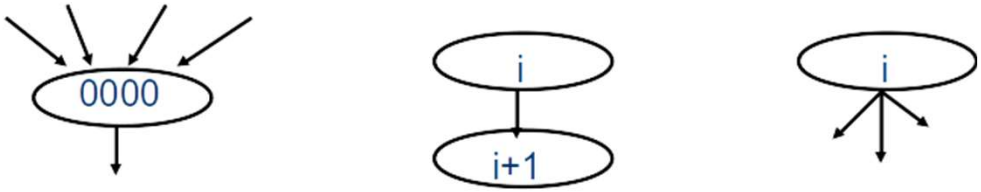
Microprograms (ctd.)

- Map arbitrary Macroinstructions (Assembly) into several microinstructions.

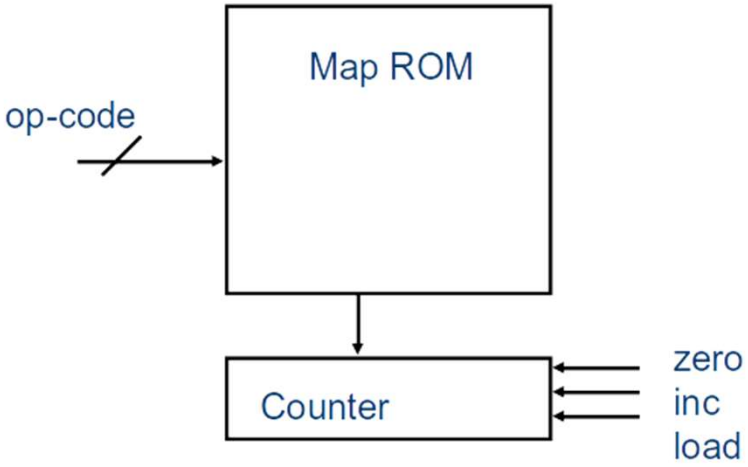


Microprograms (ctd.)

★ Map ROM for control signal

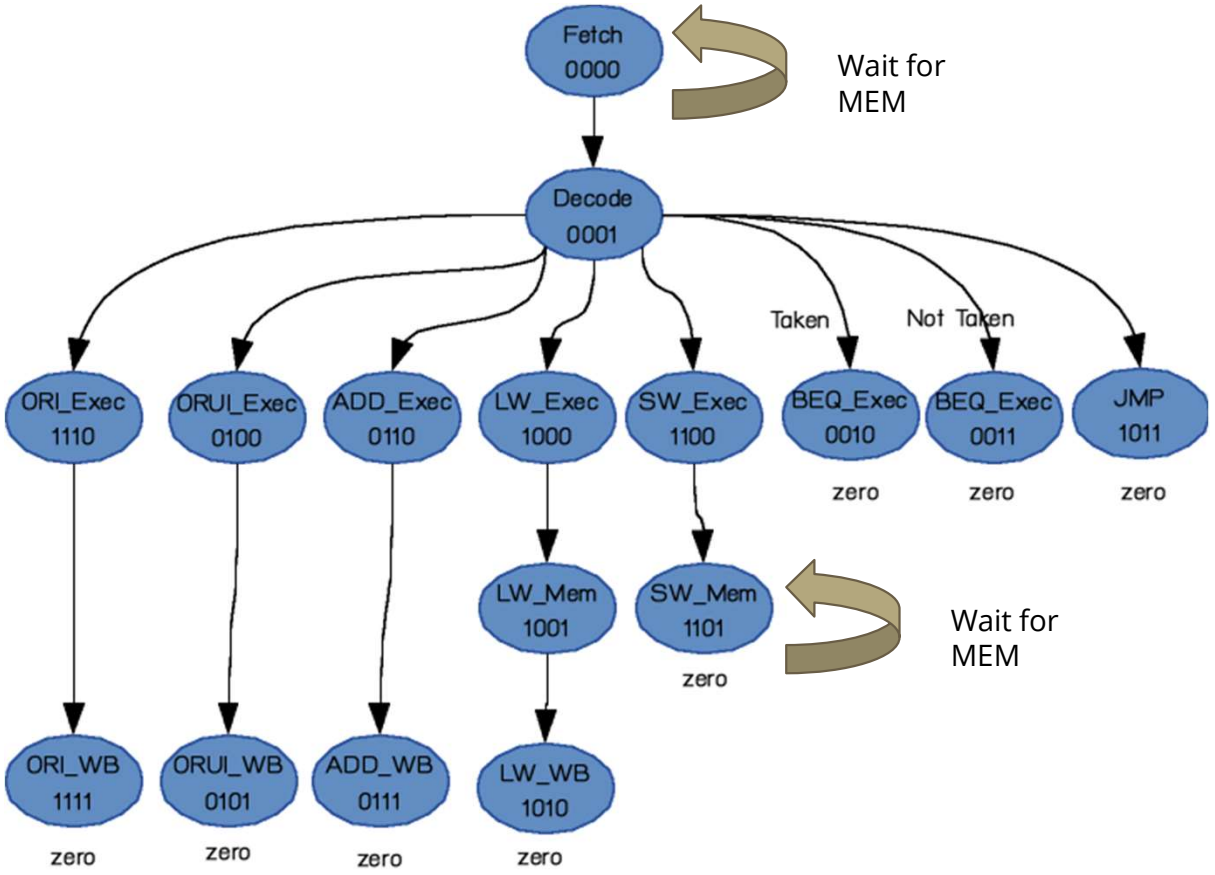


★ With appropriate hardware, we can change microprogram (Map ROM) to implement different ISA on the same hardware.



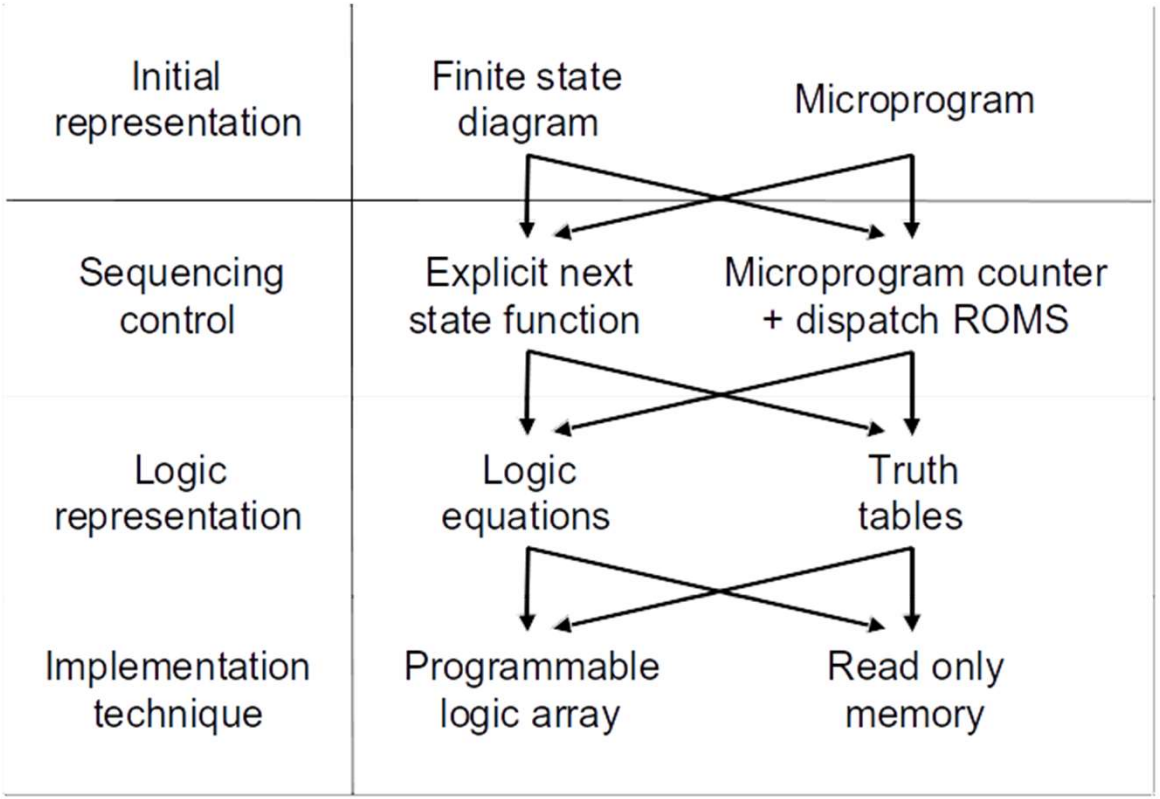
State Assignment

- ★ In reality, we may have to add wait state for the non-ideal memory



Big Picture

- ★ There are several ways to implement the processor.
- ★ For us, we will use VerilogHDL.



Summary

- ★ Disadvantages of the single cycle processor
  - Long cycle time
- ★ Advantages of multiple cycle processor
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- ★ Partition datapath into equal size chunks to minimize cycle time
- ★ Control is a finite state machine.
- ★ CISC makes control more complicated.

Notes

- We do not take exceptions into the design at the moment.

Exercises

How may cycles?

- From the given code, calculate CPU Time.
  - LW $r2, 0($r3)
  - LW $r3, 4($r3)
  - BEQ $r2, $r3, end_program #assume not taken
  - ADD $r5, $r2, $r3
  - SW $r5, 8($r3)
  - end_program:.... # end of program
- Assuming that cycle time is 40ns.

Share Components

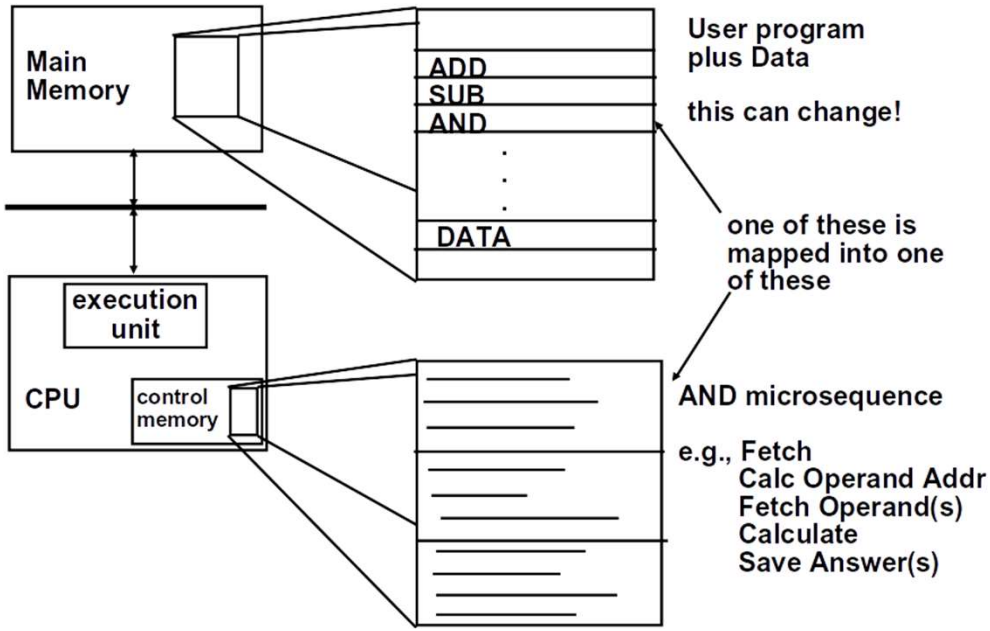
★ How to adjust the datapath so that we can share ALU with Adder (for PC)?

Interrupt and Exception

- What do we have to do to have our processor support Interrupt and Exception?

MicroInstruction

- Given that a macroinstruction always translate into several microinstructions in the processor, why don't we skip macroinsturctions and program directly in microinstructions? This should take out a translation layer in the processor.



End of Chapter 5