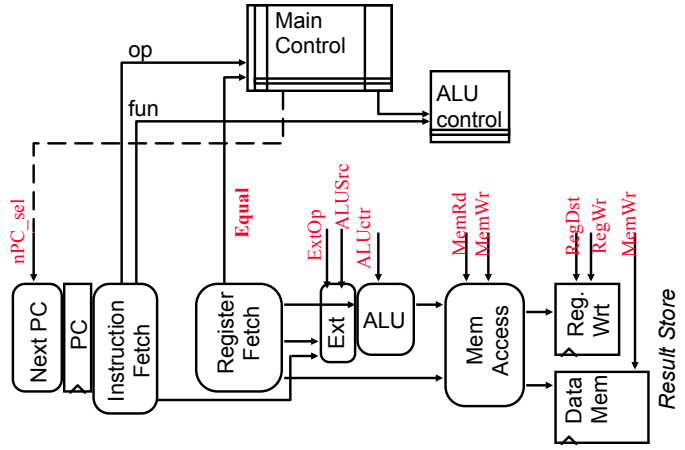
EEL-4713
Computer Architecture
Designing a Multiple-Cycle Processor

Outline of today's lecture

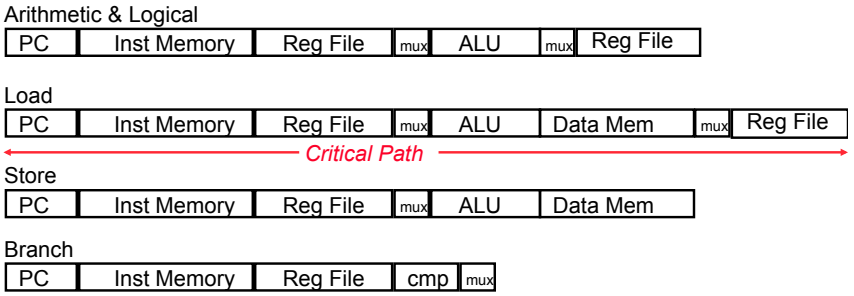
- Recap and Introduction
- Introduction to the Concept of Multiple Cycle Processor
- Multiple Cycle Implementation of R-type Instructions
- What is a Multiple Cycle Delay Path and Why is it Bad?
- Multiple Cycle Implementation of Or Immediate
- Multiple Cycle Implementation of Load and Store
- Putting it all Together

Abstract view of our single cycle processor



- looks like an FSM with PC as state

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

Drawbacks of this single cycle processor

- Long cycle time:
  - Cycle time is much longer than needed for all other instructions. Examples:
  - R-type instructions do not require data memory access
  - Jump does not require ALU operation nor data memory access
- Need for multiple functional units
  - Can't share functional units for multiple operations in the same instruction
  - E.g., instruction/data memory, adders (PC, ALU, branch target, etc.)

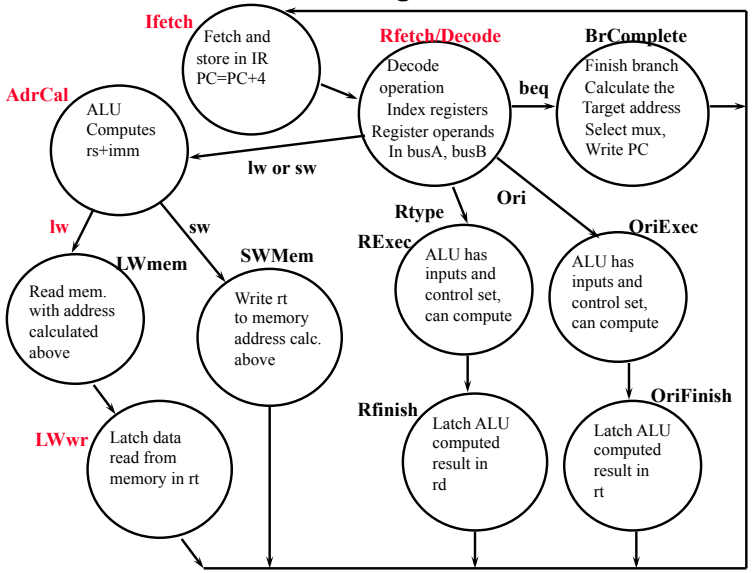
Overview of a multiple cycle implementation

- The root of the single cycle processor's problems:
  - The cycle time has to be long enough for the slowest instruction
- Solution:
  - Break the instruction into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
    - Cycle time: time it takes to execute the longest step
    - All the steps have similar length
  - This is the essence of the multiple cycle processor
- The advantages of the multiple cycle processor:
  - Cycle time is much shorter
  - Different instructions take different number of cycles to complete (for now)
    - Load takes five cycles
    - Jump only takes three cycles
  - Allows a functional unit to be used more than once per instruction

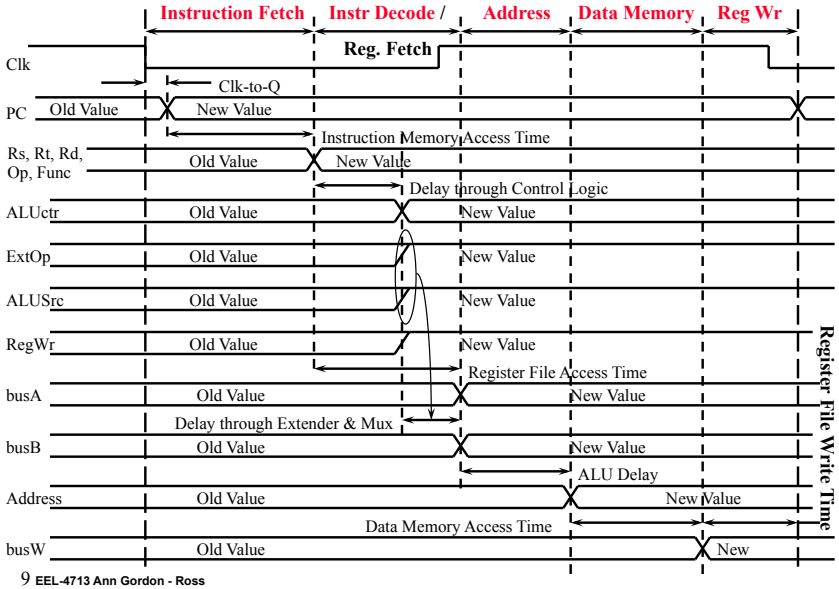
What and when:

- When designing multi-cycle implementations, you must think about:
  - *What* to do on each cycle
  - *When* results are ready for next cycle
- What to do on each cycle:
  - Always need to fetch instruction
  - Always need to decode instruction (know what to do next)
  - Next need to perform actual operation (varies from instruction to instruction)
  - E.g.:
    - Load will require address calculation, memory read, reg write
    - Branch will require comparison and PC update
    - R-type will require ALU operation, reg write

Overview: Control State Diagram

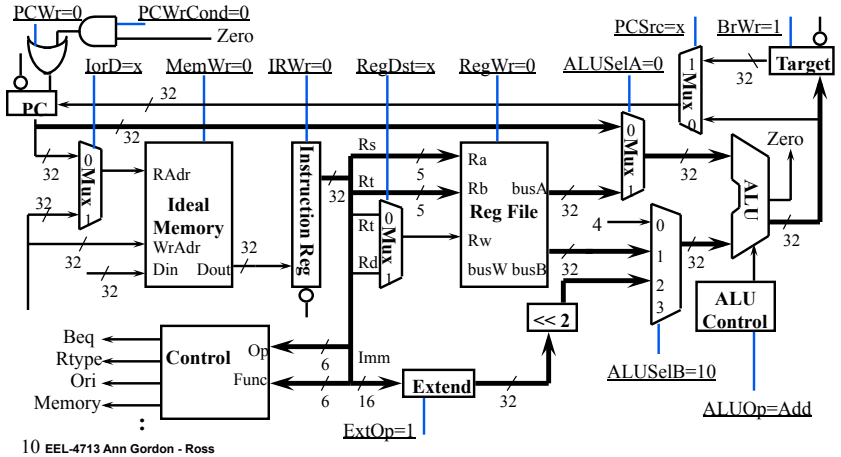


Example: the five steps of a Load instruction

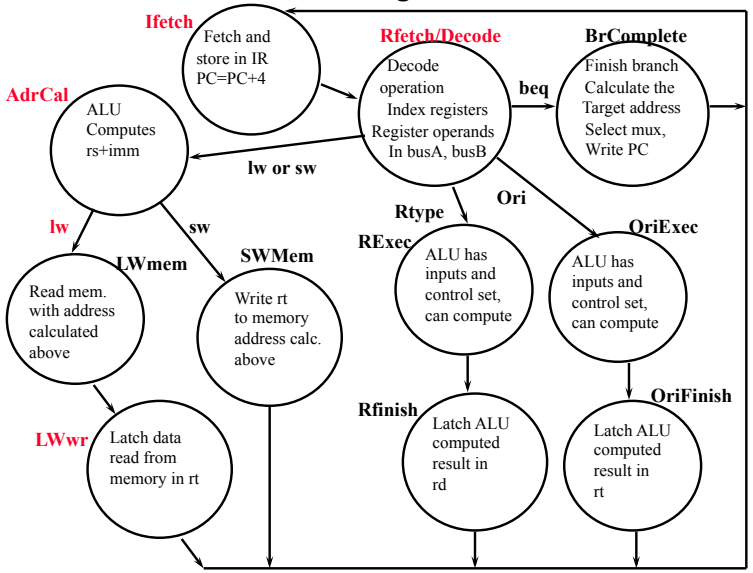


Multicycle datapath

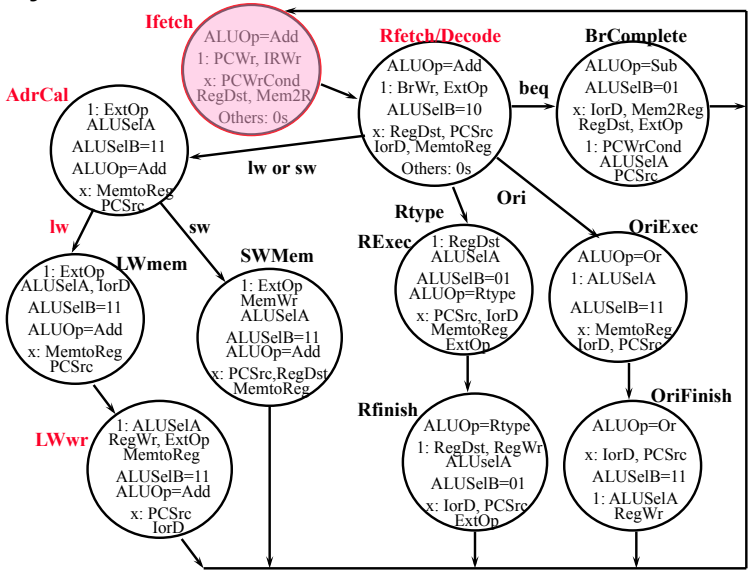
- Similar to the single-cycle datapath; use latches for instruction and branch target address
- Control signals generated for multiple clock cycles per instruction - FSM



Overview: Control State Diagram

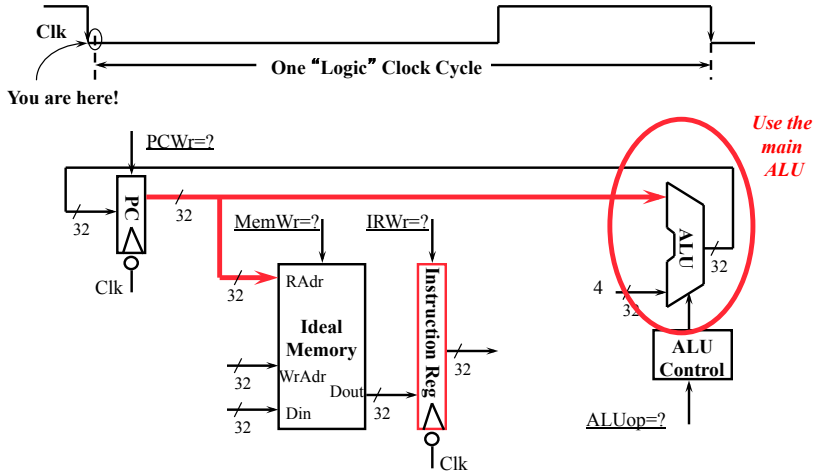


Cycle 1: Fetch



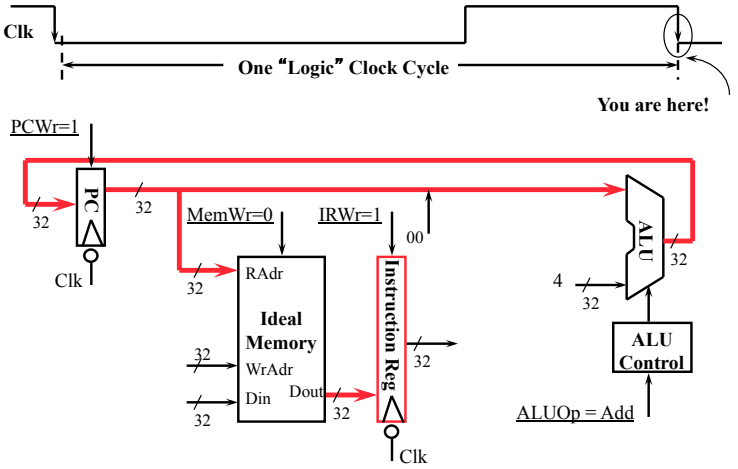
1 Instruction fetch cycle: beginning

° Every cycle begins right AFTER the clock tick:
- mem[PC] PC<31:0> + 4



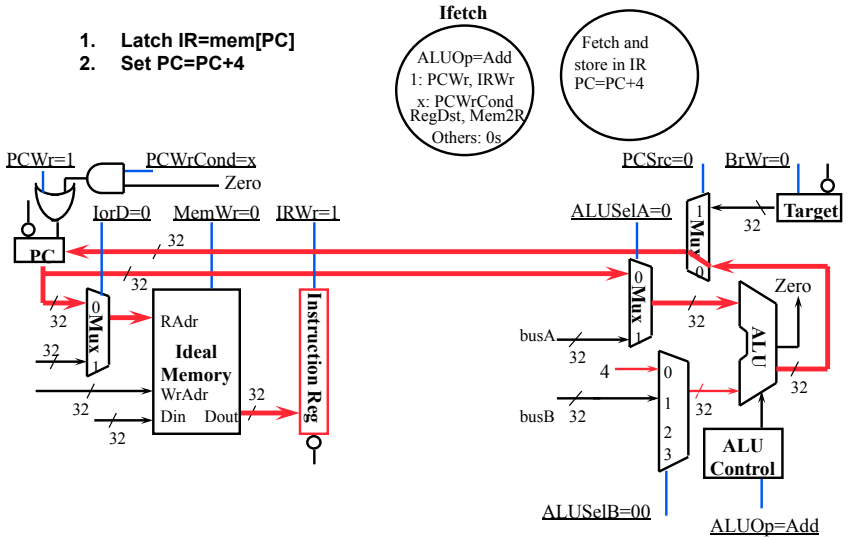
1 Instruction fetch cycle: end

° Every cycle ends AT the next clock tick (storage element updates):
- IR <-- mem[PC] PC<31:0> <-- PC<31:0> + 4

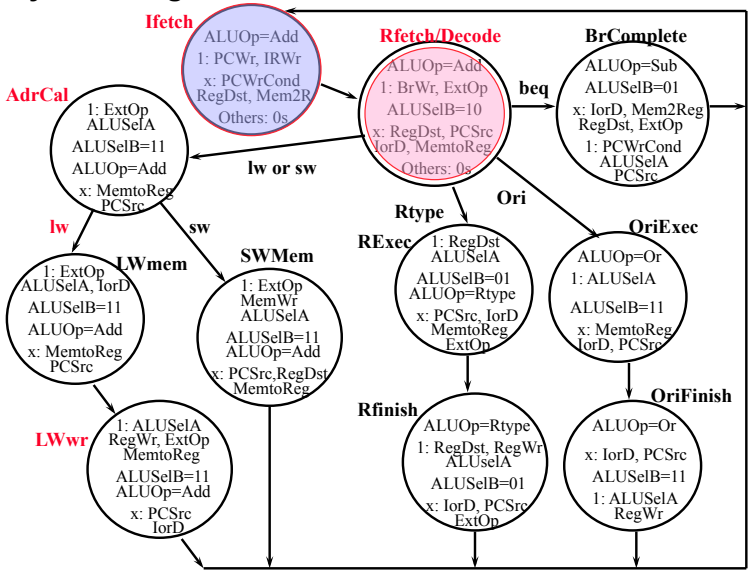


1 Instruction Fetch Cycle: Overall Picture

1. Latch IR=mem[PC]
2. Set PC=PC+4



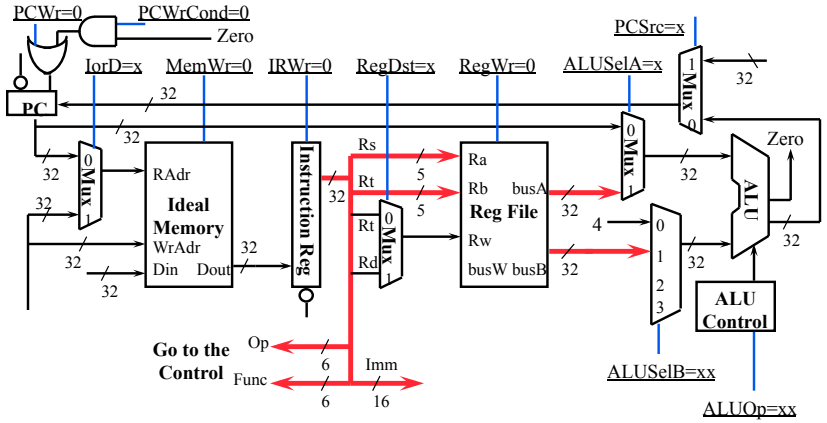
Cycle 2: Register fetch, decode



2 Register Fetch / Instruction Decode

° busA <- RegFile[rs] ; busB <- RegFile[rt] ;

° ALU can be also be used to compute branch target address (next slide) in this cycle (latch target); register compare done on next cycle

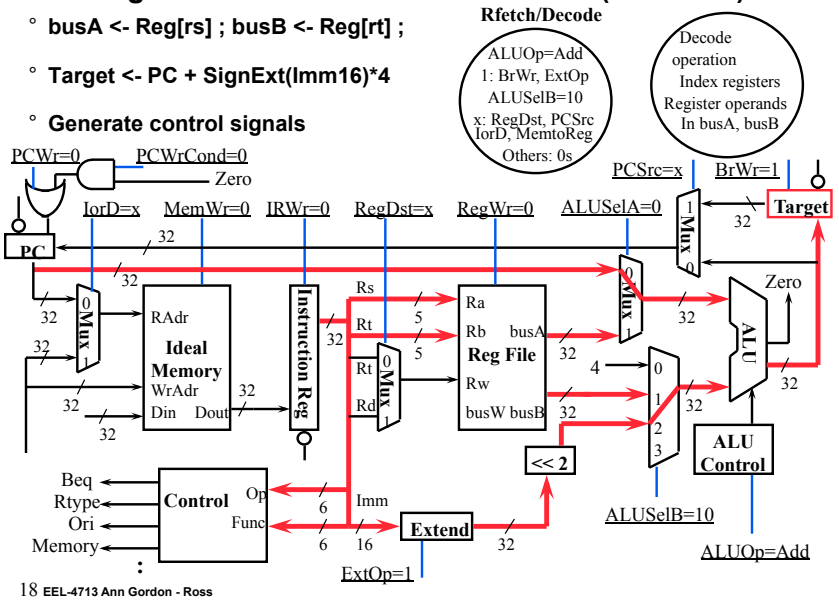


2 Register Fetch / Instruction Decode (Continue)

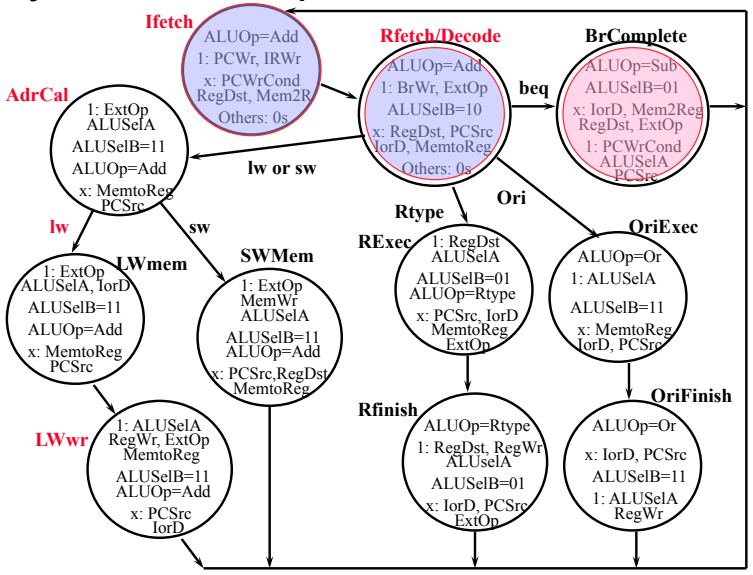
° busA <- Reg[rs] ; busB <- Reg[rt] ;

° Target <- PC + SignExt(Imm16)*4

° Generate control signals

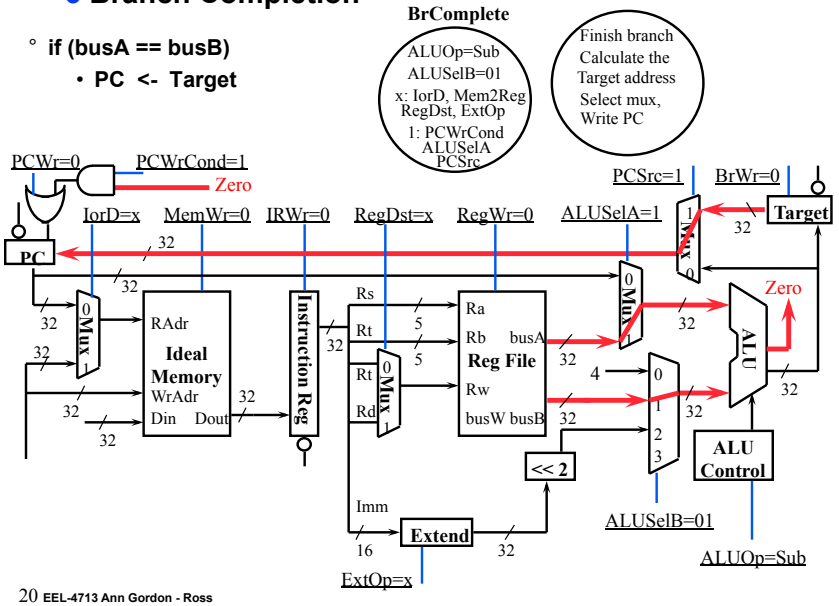


Cycle 3: Branch completion

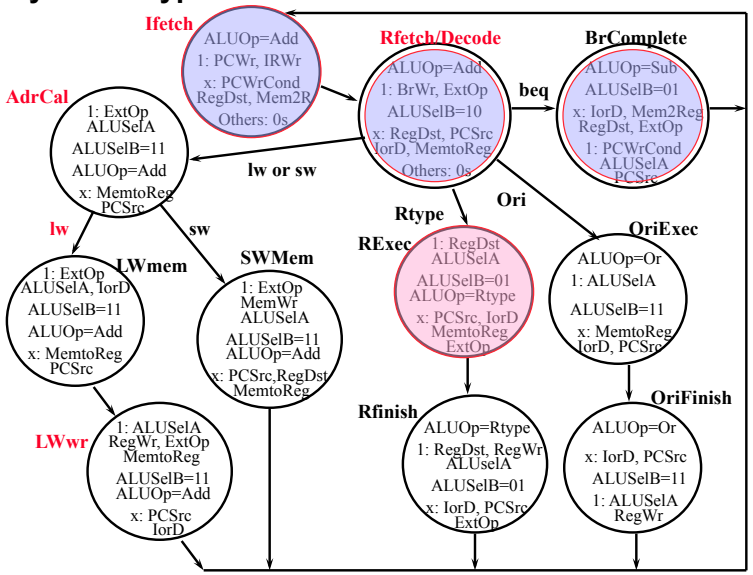


3 Branch Completion

° if (busA == busB)
  • PC <- Target

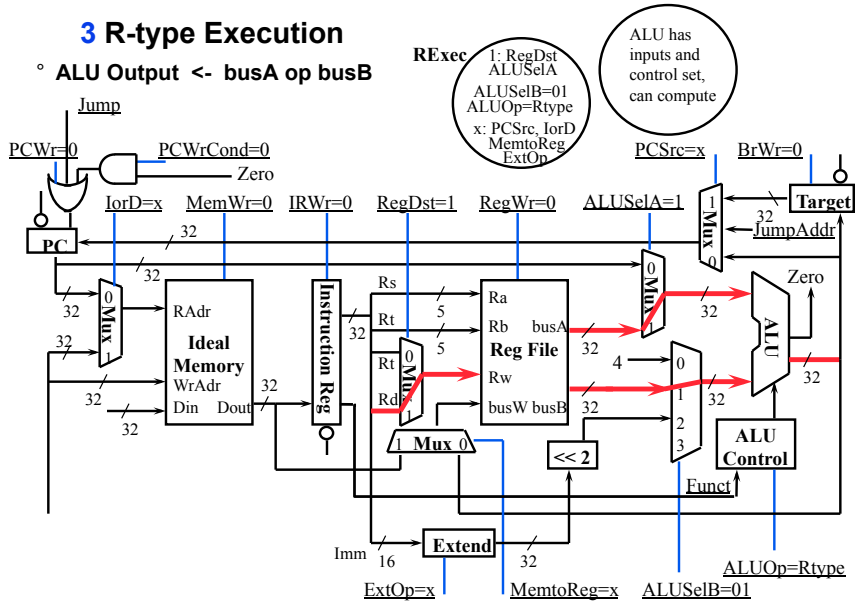


Cycle 3: Rtype execution

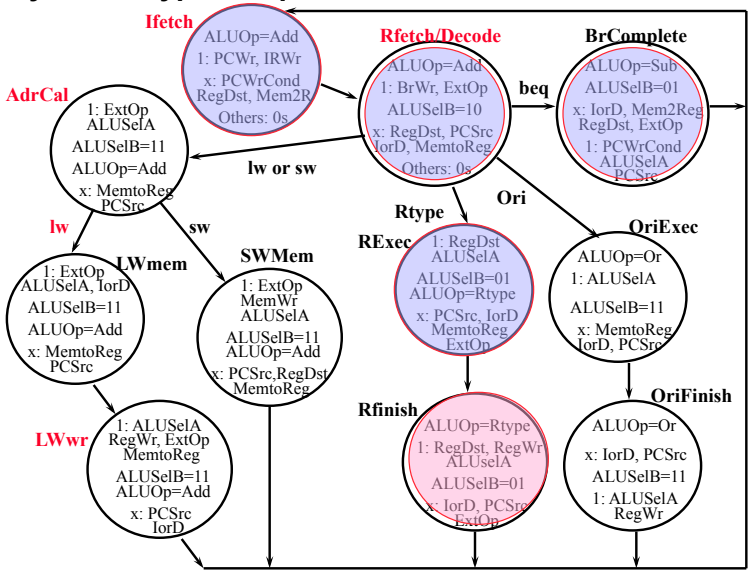


3 R-type Execution

° ALU Output <- busA op busB

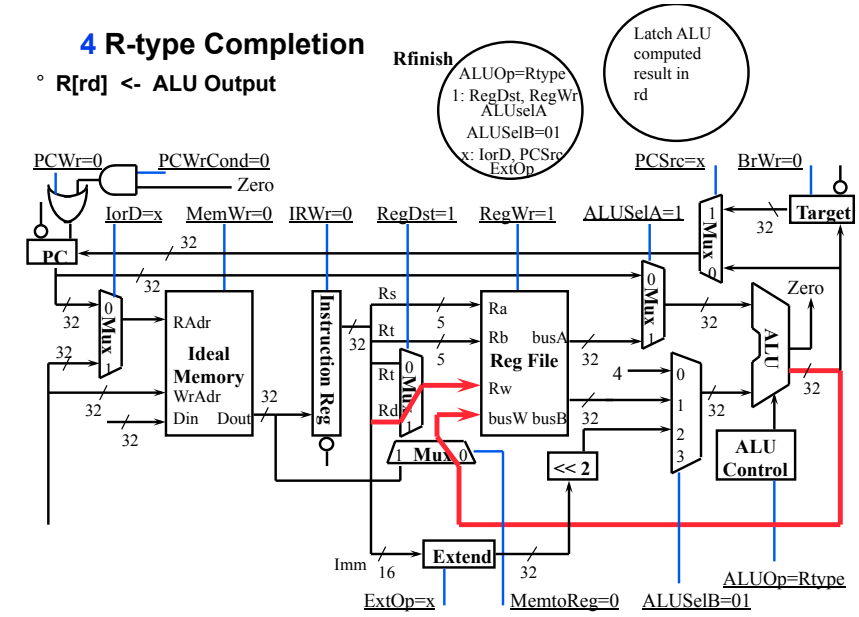


Cycle 4: Rtype completion

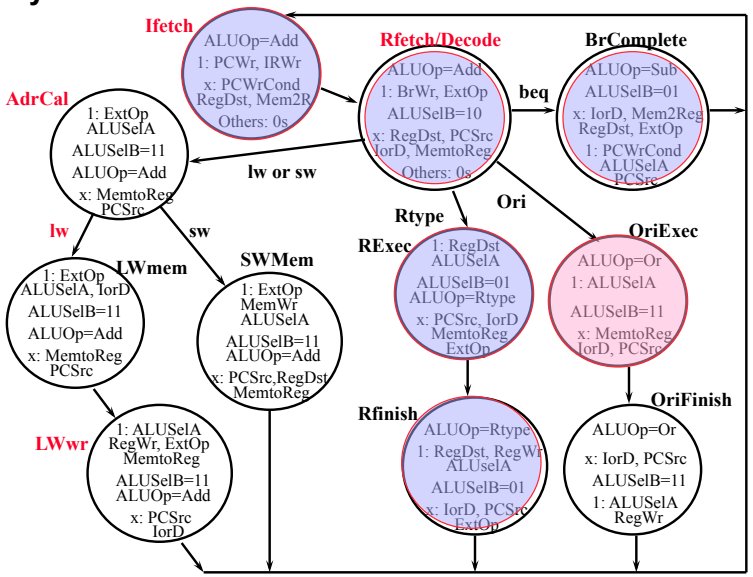


4 R-type Completion

° R[rd] <- ALU Output

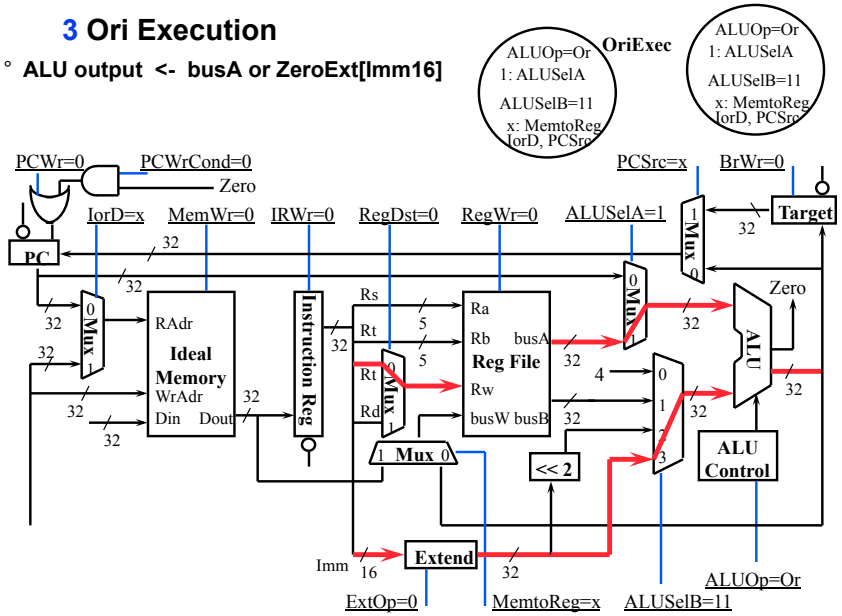


Cycle 3: Ori execution

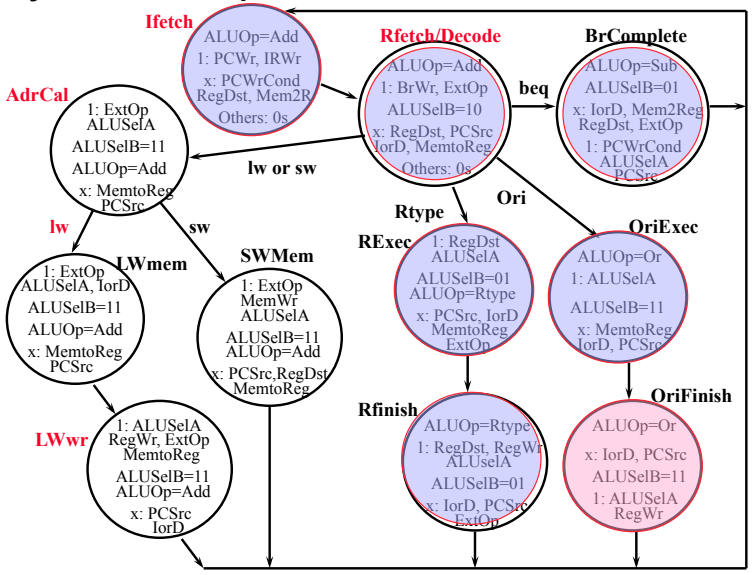


3 Ori Execution

° ALU output <- busA or ZeroExt[Imm16]

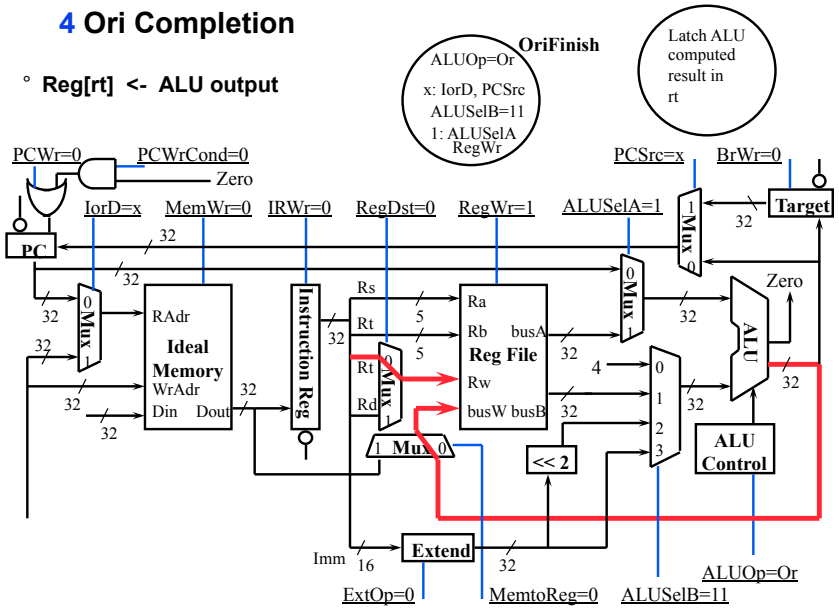


Cycle 4: Ori completion

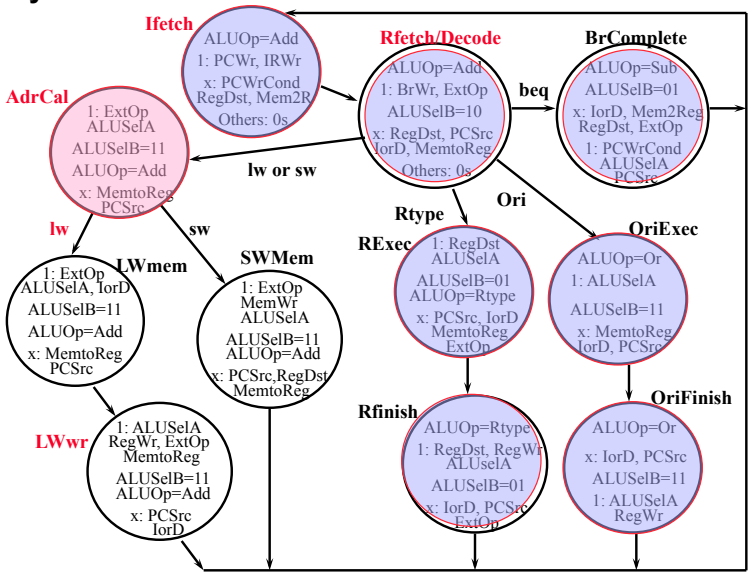


4 Ori Completion

° Reg[rt] <- ALU output

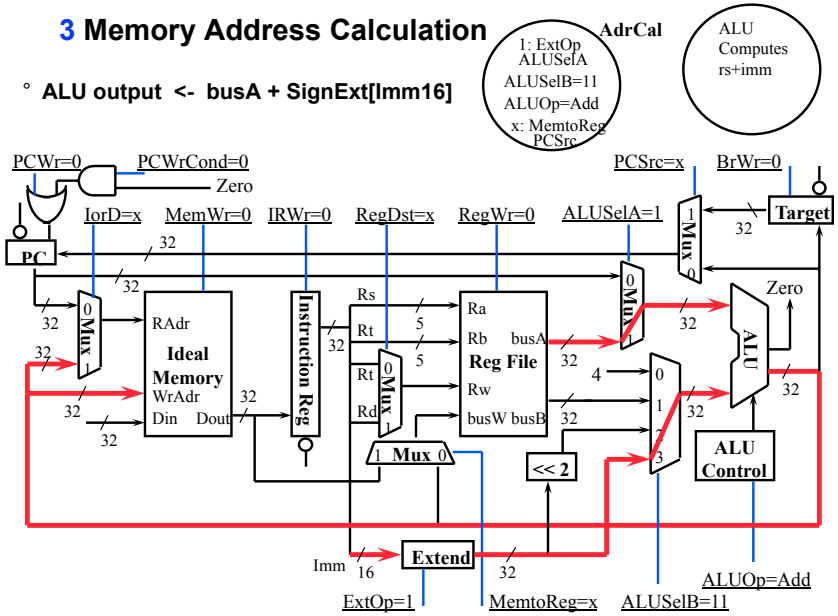


Cycle 3: Address calculation

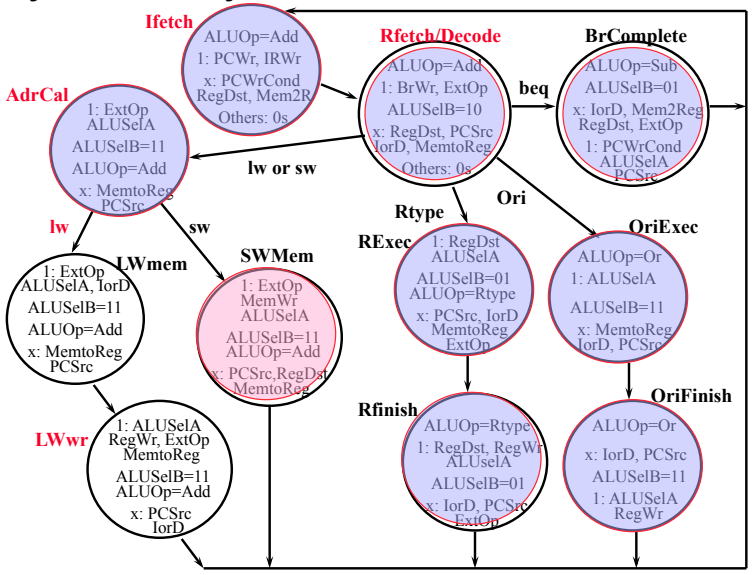


3 Memory Address Calculation

° ALU output <- busA + SignExt[Imm16]

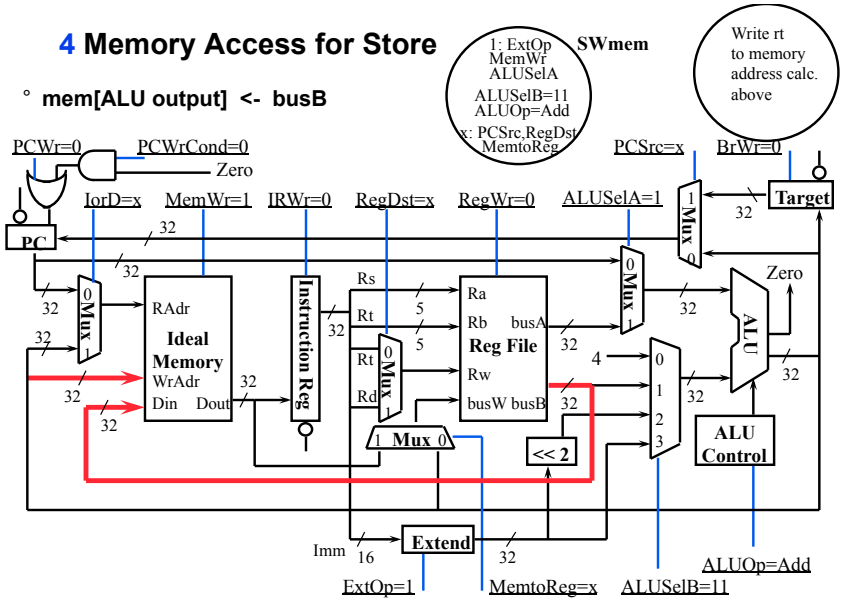


Cycle 4: Memory access, store

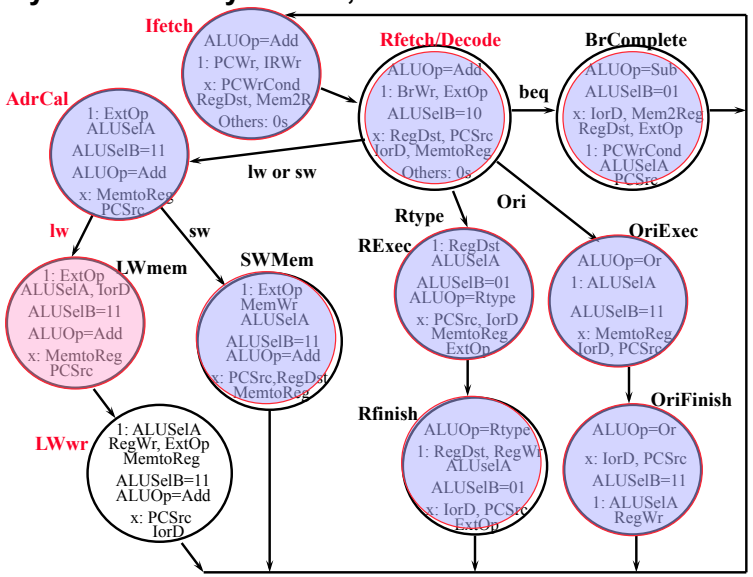


4 Memory Access for Store

° mem[ALU output] <- busB

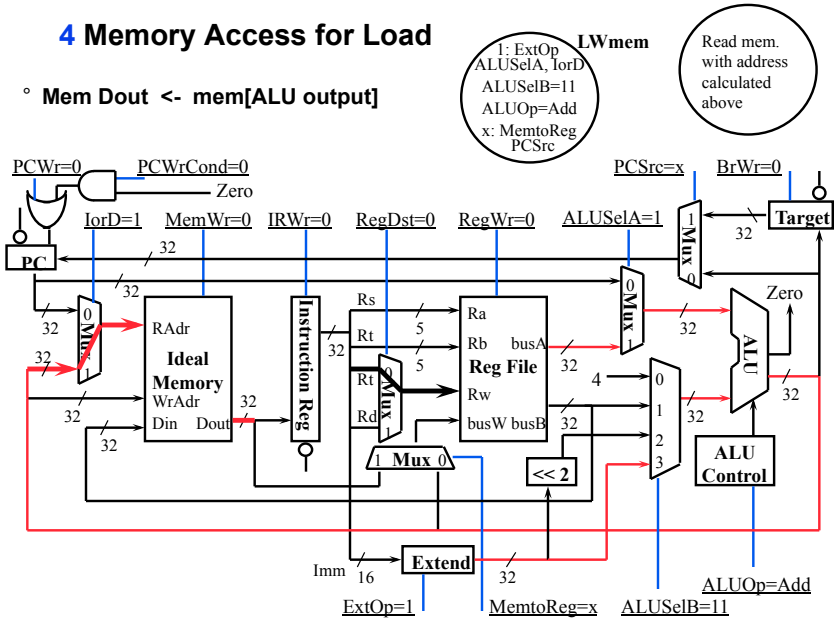


Cycle 4: Memory access, load

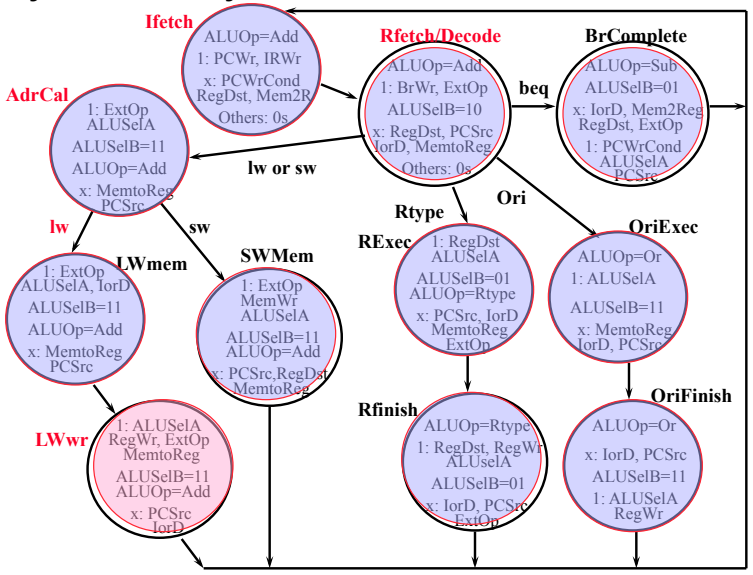


4 Memory Access for Load

- Mem Dout <- mem[ALU output]

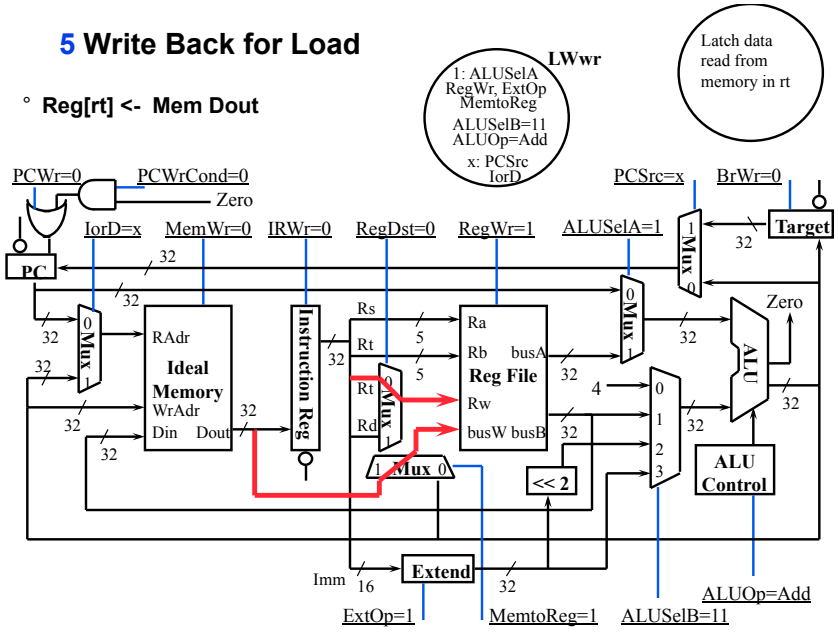


Cycle 4: Memory access, load

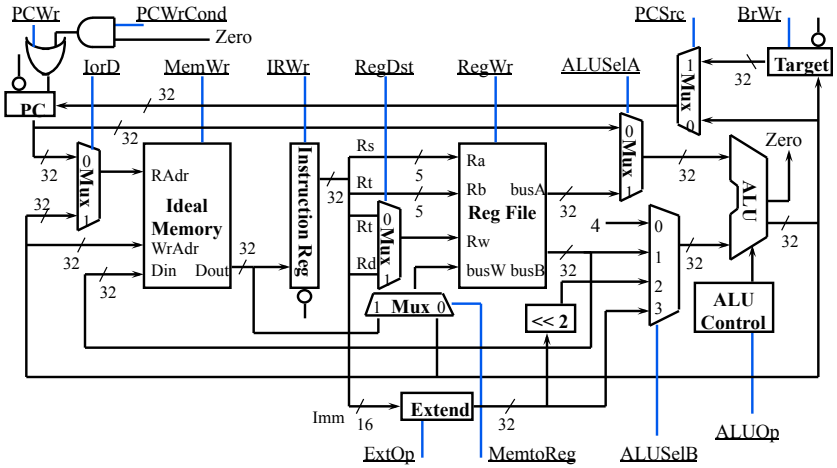


5 Write Back for Load

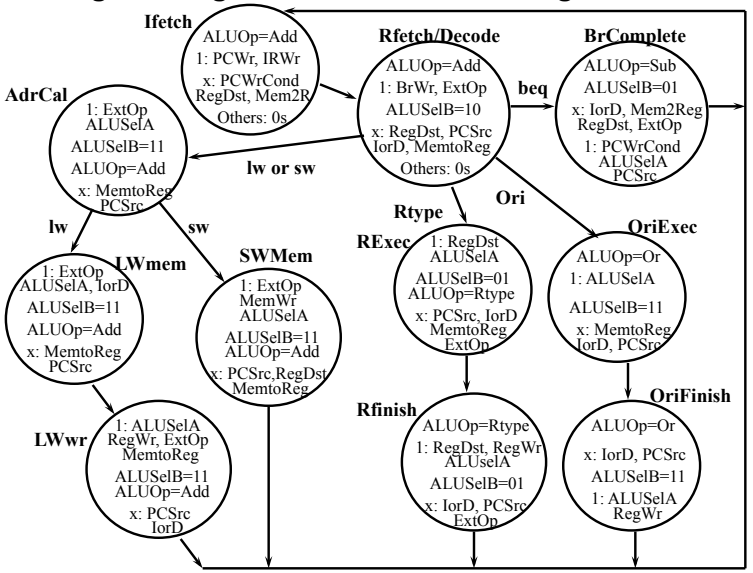
- Reg[rt] <- Mem Dout



Putting it all together: Multiple Cycle Datapath

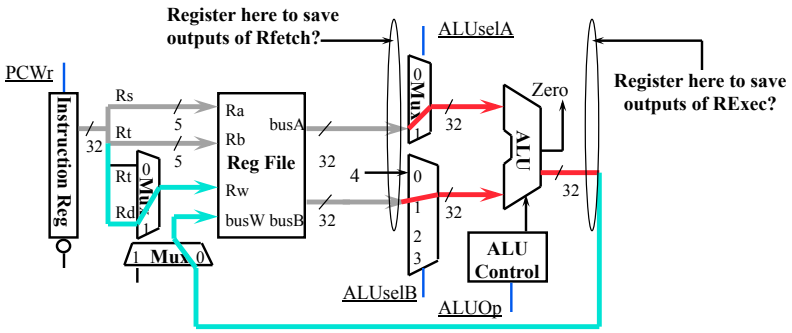


Putting it all together: Control State Diagram



Note: there is a multiple-cycle delay path

- There is no register to save the results between:
  - 2) Register Fetch: busA <- Reg[rs] ; busB <- Reg[rt]
  - 3) R-type Execution: ALU output <- busA op busB
  - 4) R-type Completion: Reg[rd] <- ALU output

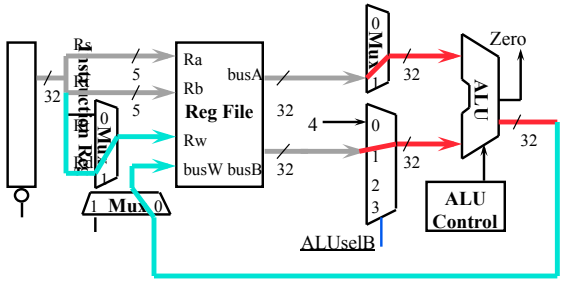


A Multiple Cycle Delay Path (Continue)

- Register is NOT needed to save the outputs of Register Fetch:
  - IRWr = 0: busA and busB will not change after Register Fetch
- Register is NOT needed to save the outputs of R-type Execution:
  - busA and busB will not change after Register Fetch
  - Control signals ALUSelA, ALUSelB, and ALUOp will not change after R-type Execution
  - Consequently ALU output will not change after R-type Execution
- In theory, you need a register to hold a signal value if:
  - (1) The signal is computed in one clock cycle and used in another.
  - (2) AND the inputs to the functional block that computes this signal *can change* before the signal is written into a state element.
- You can save a register if Cond 1 is true BUT Cond 2 is false:
  - But in practice, this will introduce a multiple cycle delay path:
    - A logic delay path that takes multiple cycles to propagate from one storage element to the next storage element

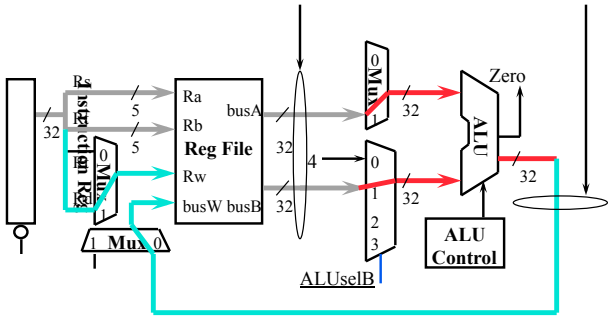
Pros and Cons of a Multiple Cycle Delay Path

- A 3-cycle path example:
  - IR (storage) -> Reg File Read -> ALU -> Reg File Write (storage)
- Advantages:
  - Register savings
  - We can share time among cycles:
    - If ALU takes longer than one cycle, still OK as long as the entire path takes less than 3 cycles to finish



Pros and Cons of a Multiple Cycle Delay Path (Continue)

- Disadvantage:
  - Static timing analyzer, which ONLY looks at delay between two storage elements, will report this as a timing violation
  - You have to ignore the static timing analyzer's warnings



Summary

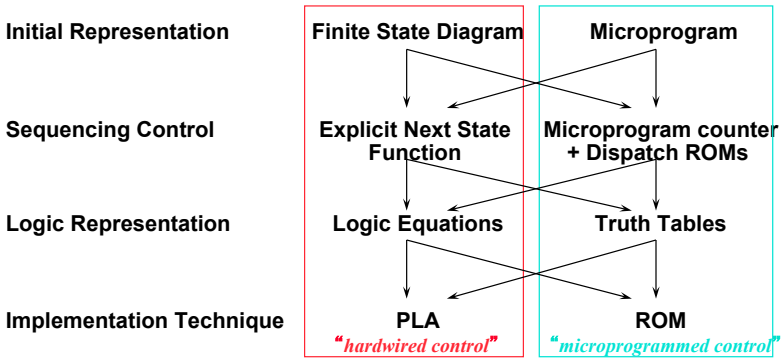
- Disadvantages of the Single Cycle Processor
  - Long cycle time
  - Cycle time is too long for all instructions except the Load
- Multiple Cycle Processor:
  - Divide the instructions into smaller steps
  - Execute each step (instead of the entire instruction) in one cycle
- Do NOT confuse Multiple Cycle Processor with multiple cycle delay path
  - Multiple Cycle Processor executes each instruction in multiple clock cycles
  - Multiple Cycle Delay Path: a combinational logic path between two storage elements that takes more than one clock cycle to complete
- It is possible (desirable) to build a MC Processor without MCDP:
  - Use a register to save a signal's value whenever a signal is generated in one clock cycle and used in another cycle later

Control logic

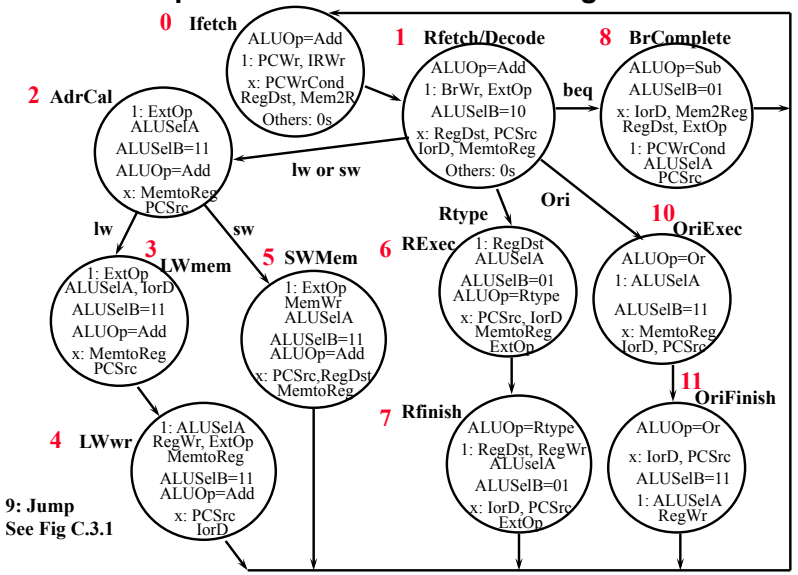
- Review of Finite State Machine (FSM) control
- From Finite State Diagrams to Microprogramming

Overview

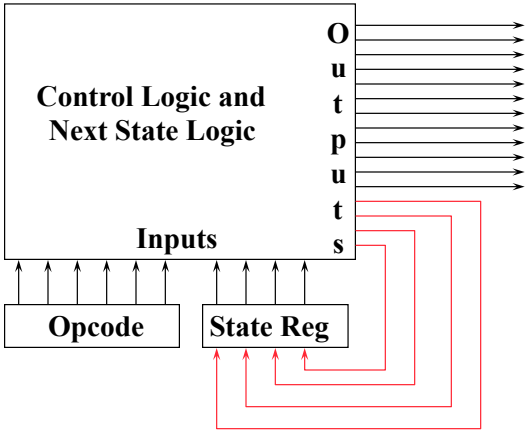
° Control may be designed using one of several initial representations. The choice of sequence control, and how logic is represented, can then be determined independently; the control can then be implemented with one of several methods using a structured logic technique.



Initial Representation: Finite State Diagram



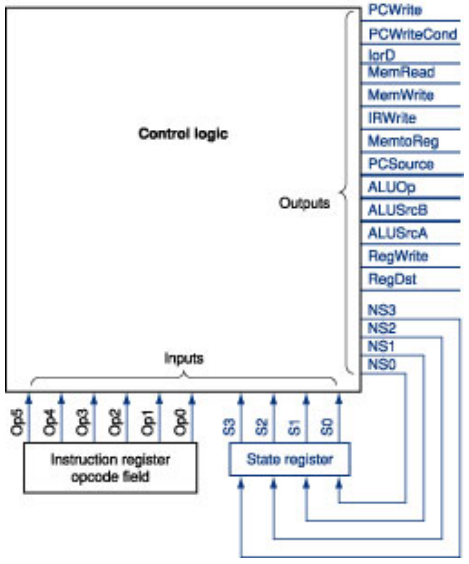
Sequencing Control: Explicit Next State Function



Multicycle Datapath

° Next state number is encoded just like datapath controls

Interface in detail



Logic Representation: Logic Equations

° Next state from current state
- State 0 -> State1
- State 1 -> S2, S6, S8, S10
- State 2 -> S3, S5
- State 3 -> State 4
- State 4 ->State 0
- State 5 -> State 0
- State 6 -> State 7
- State 7 -> State 0
- State 8 -> State 0
- State 9-> State 0
- State 10 -> State 11
- State 11 -> State 0

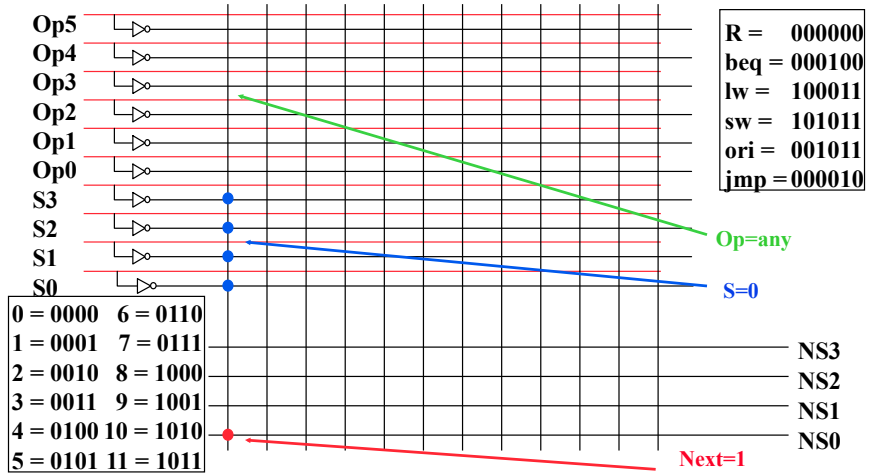
°Alternatively,
prior state & condition

S4, S5, S7, S8, S9, S11 -> State0
State 0_____ -> State 1
State 1 & op = lw|sw -> State 2
State2 & op = lw ____ -> State 3
State 3 _____ -> State 4
State2 & op = sw ____ -> State 5
State 1 & op = R-type -> State 6
State 6 _____-> State 7
State 1 & op = beq___ -> State 8
State2 & op = jmp ___-> State 9
State 1& op = ORi__ -> State 10
State 10 _____ -> State 11

See Fig. C.3.3

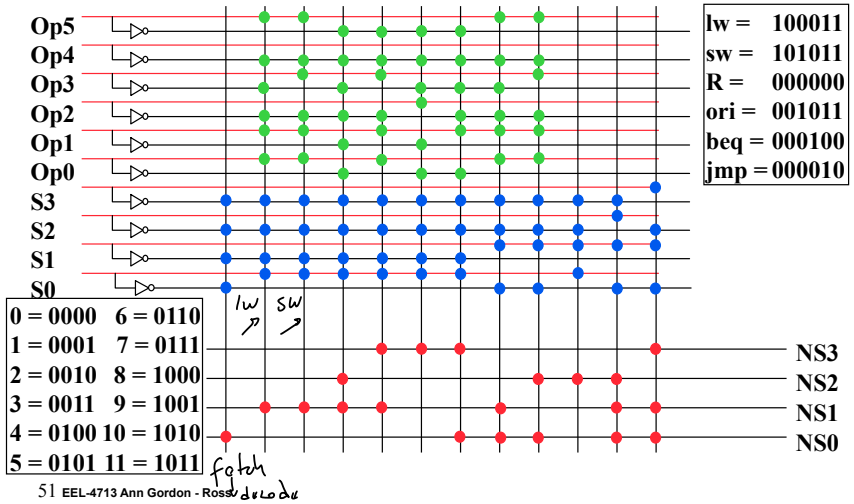
Implementation Technique: Programmed Logic Arrays

° Each output line: the logical OR of logical AND of input lines or their complement; AND minterms specified in top AND plane, OR sums specified in bottom OR plane



Implementation Technique: Programmed Logic Arrays

° Each output line the logical OR of logical AND of input lines or their complement: AND minterms specified in top AND plane, OR sums specified in bottom OR plane



Multicycle Control

° Given numbers assigned to FSM, can in turn determine next state as function of the inputs and current state

° Can turn these into Boolean equations for each bit of the next state lines
- Implement easily using PLA (programmable logic array) or ROM storing truth tables
- See Figs. C.3.6 and C.3.8 for tables showing outputs and next state as function of current state and opcode

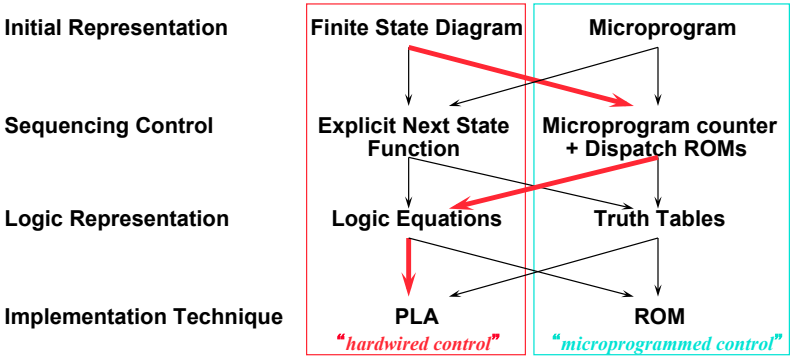
° What if many more states, many more conditions?
- State machine gets too large; very large ROMs/PLAs

° What if need to add a state?
- May need to increase address for ROM, number of inputs for PLA gates

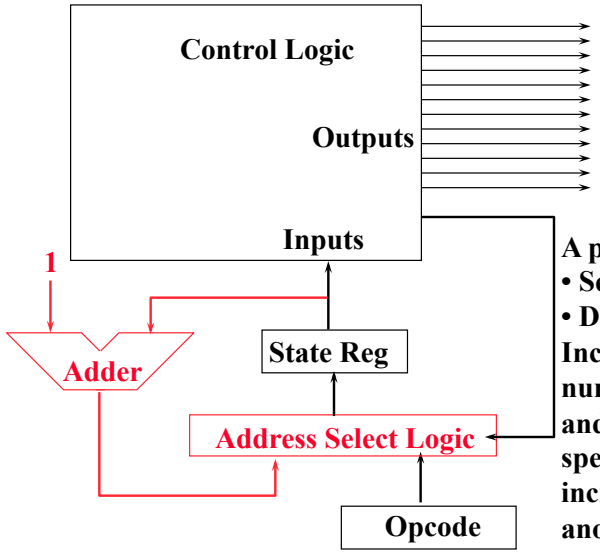
° Or just implement FSM in VHDL

Next Iteration: Using Sequencer for Next State

- Before: Explicit Next State; Next try variation 1 step from right hand side
- Few sequential states in small FSM: suppose added floating point?
- Still need to go to non-sequential states: e.g., state 1 => 2, 6, 8, 10



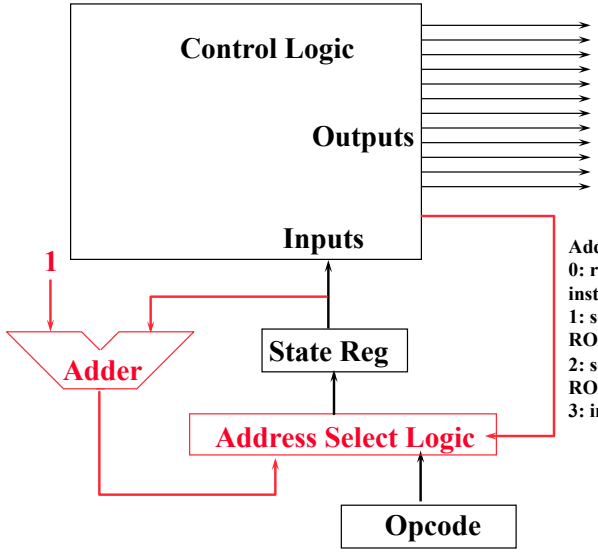
Sequencer-based control unit



Multicycle Datapath

A processor on its own
• Set state to 0
• Dispatch (state 1 & 2)
Incremented state number ("microPC") and "micro branches" specify whether to increment or select another state based on opcode

Sequencer-based control unit



Multicycle Datapath

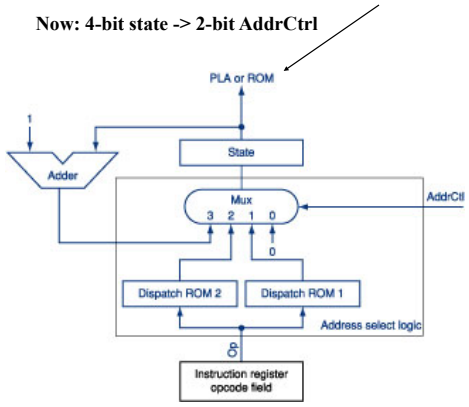
AddrCtrl control for select logic
0: reset state register (next instruction)
1: select next using next-state ROM for state 1
2: select next using next-state ROM for state 2
3: increment state

Fig. C.4.2

Sequencer block diagram

Before: 6bit opcode + 4-bit state -> 4-bit NS

Now: 4-bit state -> 2-bit AddrCtrl

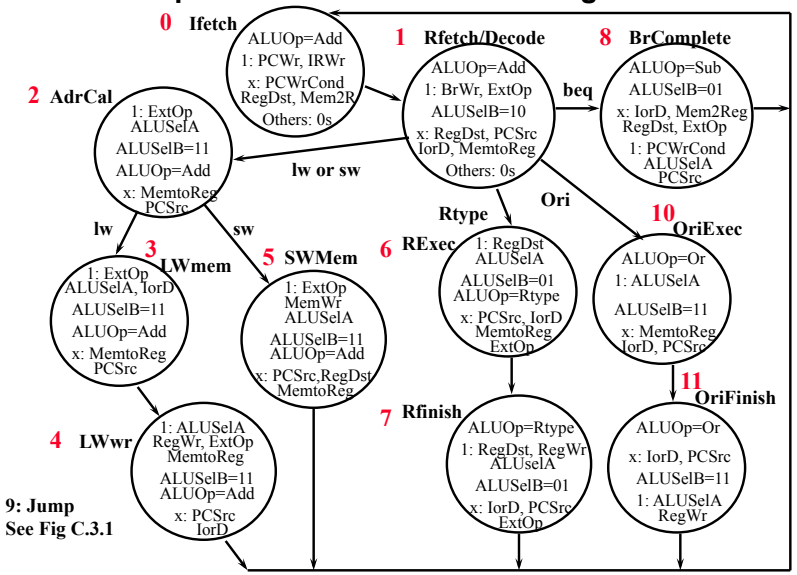


0000 – AddrCtrl=3 (fetch)
0001 – AddrCtrl=1 (decode)
0010 – AddrCtrl=2 (lw/sw)
0011 – AddrCtrl=3 (lw)
0100 – AddrCtrl=0 (lw)
0101 – AddrCtrl=0 (sw)
0110 – AddrCtrl=3 (r-type)
0111 – AddrCtrl=0 (r-type)
1000 – AddrCtrl=0 (branch)
1010 – AddrCtrl=3 (ori)
1011 – AddrCtrl=0 (ori)

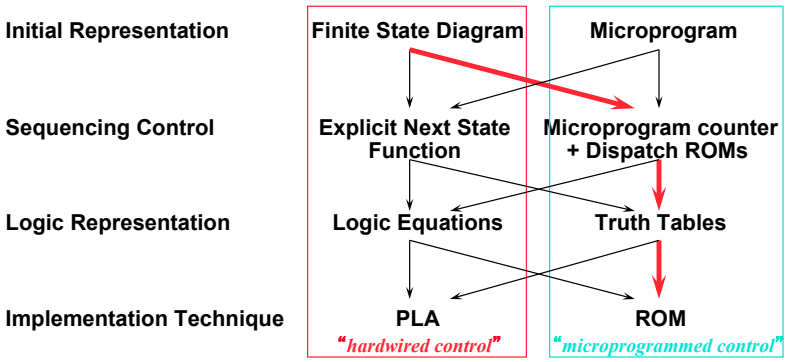
Dispatch ROM 1 (Indexed by opcode)
lw -> 0010
sw -> 0010
R-type -> 0110
ori -> 1010
branch -> 1000

ROM2:
lw -> 0011
sw -> 0101

Initial Representation: Finite State Diagram



Next Iteration: Using Microprogram for Representation



° ROM can be thought of as a sequence of control words

° Control word can be thought of as instruction: "microinstruction"

Microprogramming

° Control is the hard part of processor design
  - ° Datapath is fairly regular and well-organized
  - ° Memory is highly regular
  - ° Control is irregular and global

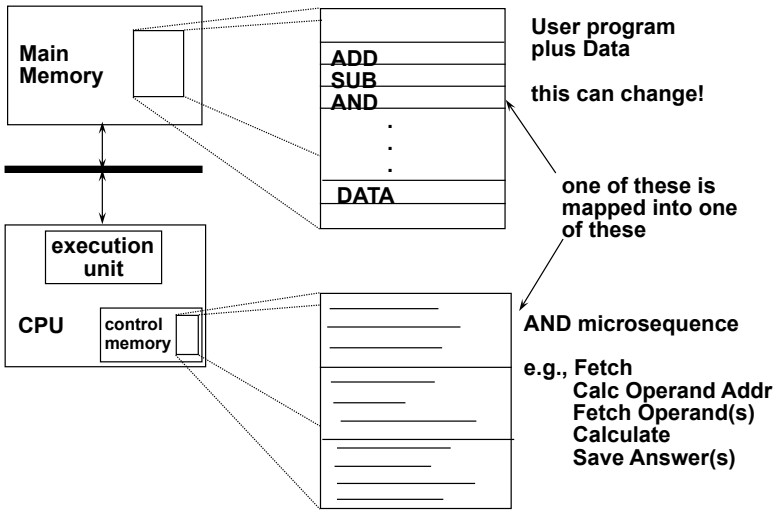
Microprogramming:

-- A particular strategy for implementing the control unit of a processor by "programming" at the level of register transfer operations

Microarchitecture:

-- Logical structure and functional capabilities of the hardware as seen by the microprogrammer

Macroinstruction Interpretation



Microprogramming Pros and Cons

- Ease of design
- Flexibility
  - Easy to adapt to changes in organization, timing, technology
  - Can make changes late in design cycle, or even in the field
- Can implement very powerful instruction sets (just more control memory)
- Generality
  - Can implement multiple instruction sets on same machine.
  - Can tailor instruction set to application.
- Compatibility
  - Many organizations, same instruction set
- Costly to implement
- Slow

Summary: Multicycle Control

- Microprogramming and hardwired control have many similarities, perhaps biggest difference is initial representation and ease of change of implementation, with ROM generally being easier than PLA