COMP303 - Computer Architecture

Midterm I

Alp Bayrakci

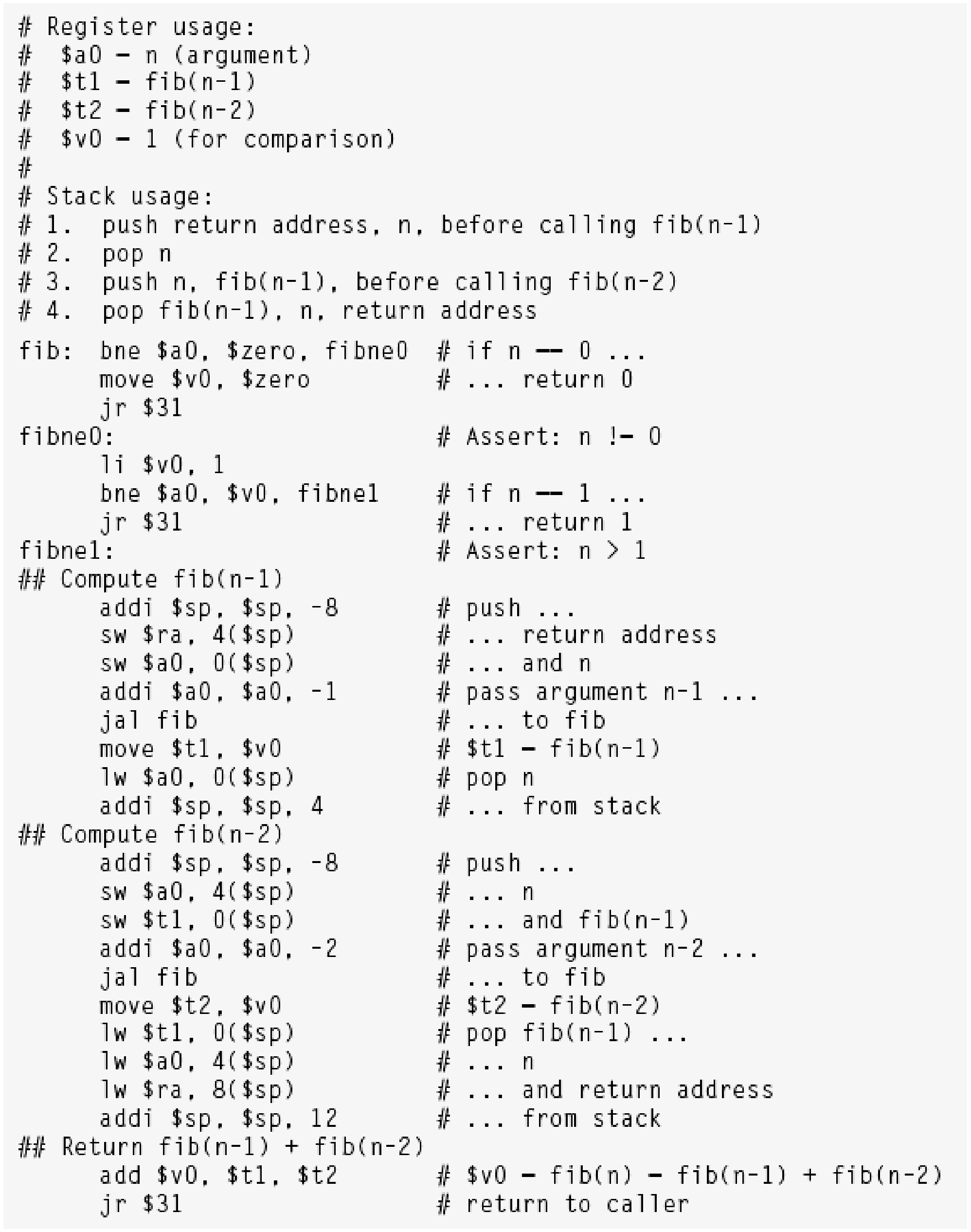
Assembly Code Example 1

```
fact:
    addi  $sp, $sp, -8       # adjust stack for 2 items
    sw    $ra, 4($sp)        # save the return address
    sw    $a0, 0($sp)        # save the argument n

    slti  $t0, $a0, 1        # test for n<1
    beq   $t0, $zero, L1     # if n>=1, goto L1
    addi  $v0, $zero, 1      # return 1
    addi  $sp, $sp, 8        # pop 2 items off stack
    jr    $ra                # return to after jal
L1:
    addi  $a0, $a0, -1       # n>=1: argument gets (n-1)
    jal   fact               # call fact with (n-1)
    lw    $a0, 0($sp)        # return from jal: restore argument n
    lw    $ra, 4($sp)        # restore the return address
    addi  $sp, $sp, 8        # adjust stack pointer to pop 2 items

    mul   $v0, $a0, $v0      # return n*fact(n-1)
    jr    $ra
```

Assembly Code Example 2



```
# Register usage:
#  $a0 - n (argument)
#  $t1 - fib(n-1)
#  $t2 - fib(n-2)
#  $v0 - 1 (for comparison)
#
# Stack usage:
# 1.  push return address, n, before calling fib(n-1)
# 2.  pop n
# 3.  push n, fib(n-1), before calling fib(n-2)
# 4.  pop fib(n-1), n, return address

fib:  bne $a0, $zero, fibne0  # if n -- 0 ...
      move $v0, $zero         # ... return 0
      jr $31
fibne0:                       # Assert: n !- 0
      li $v0, 1
      bne $a0, $v0, fibne1    # if n -- 1 ...
      jr $31                  # ... return 1
fibne1:                       # Assert: n > 1
## Compute fib(n-1)
      addi $sp, $sp, -8       # push ...
      sw $ra, 4($sp)          # ... return address
      sw $a0, 0($sp)          # ... and n
      addi $a0, $a0, -1       # pass argument n-1 ...
      jal fib                 # ... to fib
      move $t1, $v0           # $t1 - fib(n-1)
      lw $a0, 0($sp)          # pop n
      addi $sp, $sp, 4        # ... from stack
## Compute fib(n-2)
      addi $sp, $sp, -8       # push ...
      sw $a0, 4($sp)          # ... n
      sw $t1, 0($sp)          # ... and fib(n-1)
      addi $a0, $a0, -2       # pass argument n-2 ...
      jal fib                 # ... to fib
      move $t2, $v0           # $t2 - fib(n-2)
      lw $t1, 0($sp)          # pop fib(n-1) ...
      lw $a0, 4($sp)          # ... n
      lw $ra, 8($sp)          # ... and return address
      addi $sp, $sp, 12       # ... from stack
## Return fib(n-1) + fib(n-2)
      add $v0, $t1, $t2       # $v0 - fib(n) - fib(n-1) + fib(n-2)
      jr $31                  # return to caller
```

Computer Performance

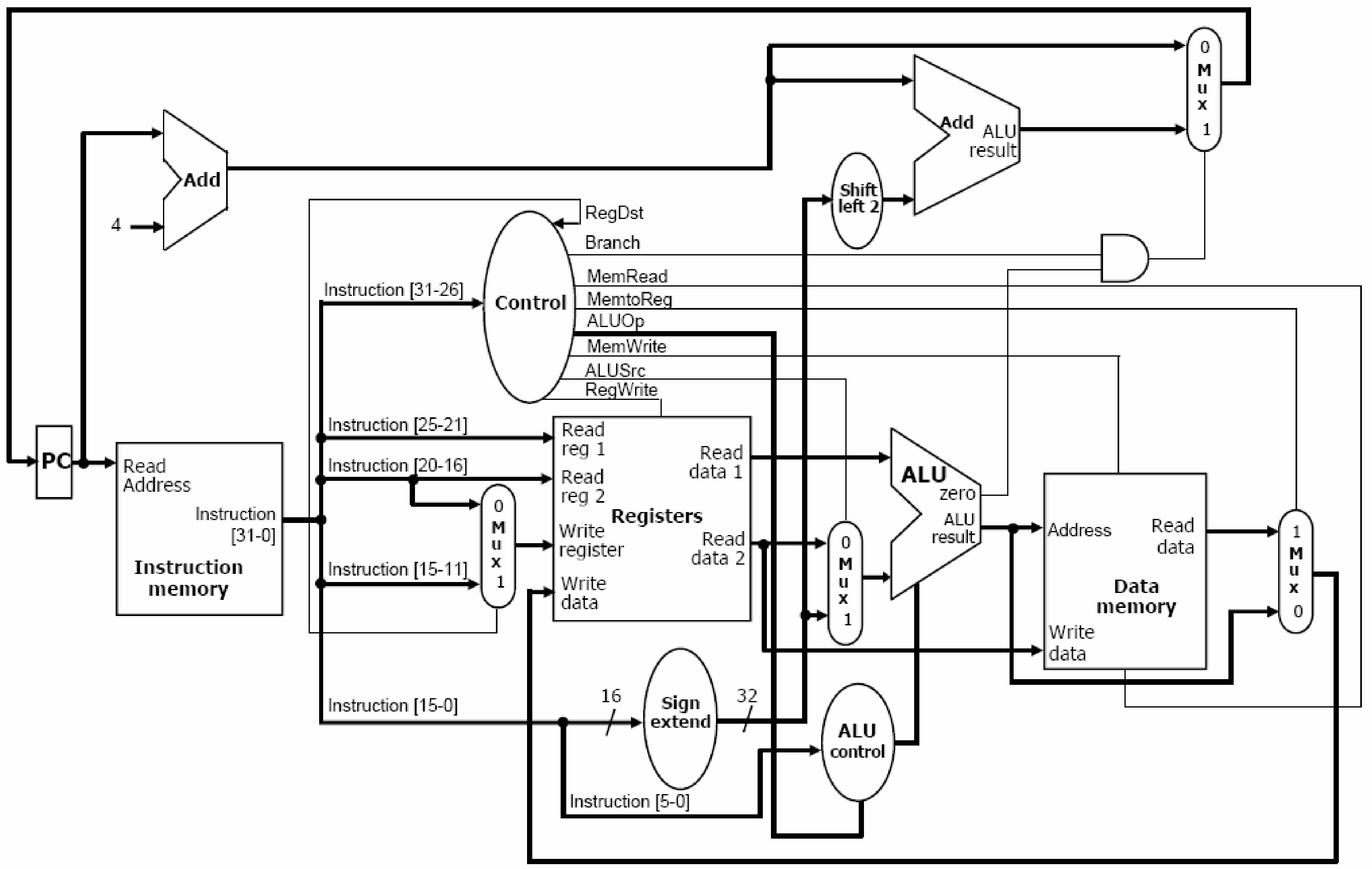
Consider two different implementations, M1 and M2, of the same instruction set. **M1** is a **single-cycle** implementation with a clock rate of **200 MHz** and **M2** is a **multiple-clock-cycle** version which operates with a **900 MHz** clock. Assume that multicycle datapath discussed in class is used for M2 to determine the number of clock cycles for each instruction type. Given a program with the mix **20% loads, 10% stores, 50% ALU operations, 20% jumps and branches**, calculate which machine is faster executing that program and by how much? Assume that jump and branch instructions take same number of cycles.

ExecTime1 = N x 1/200 MHz

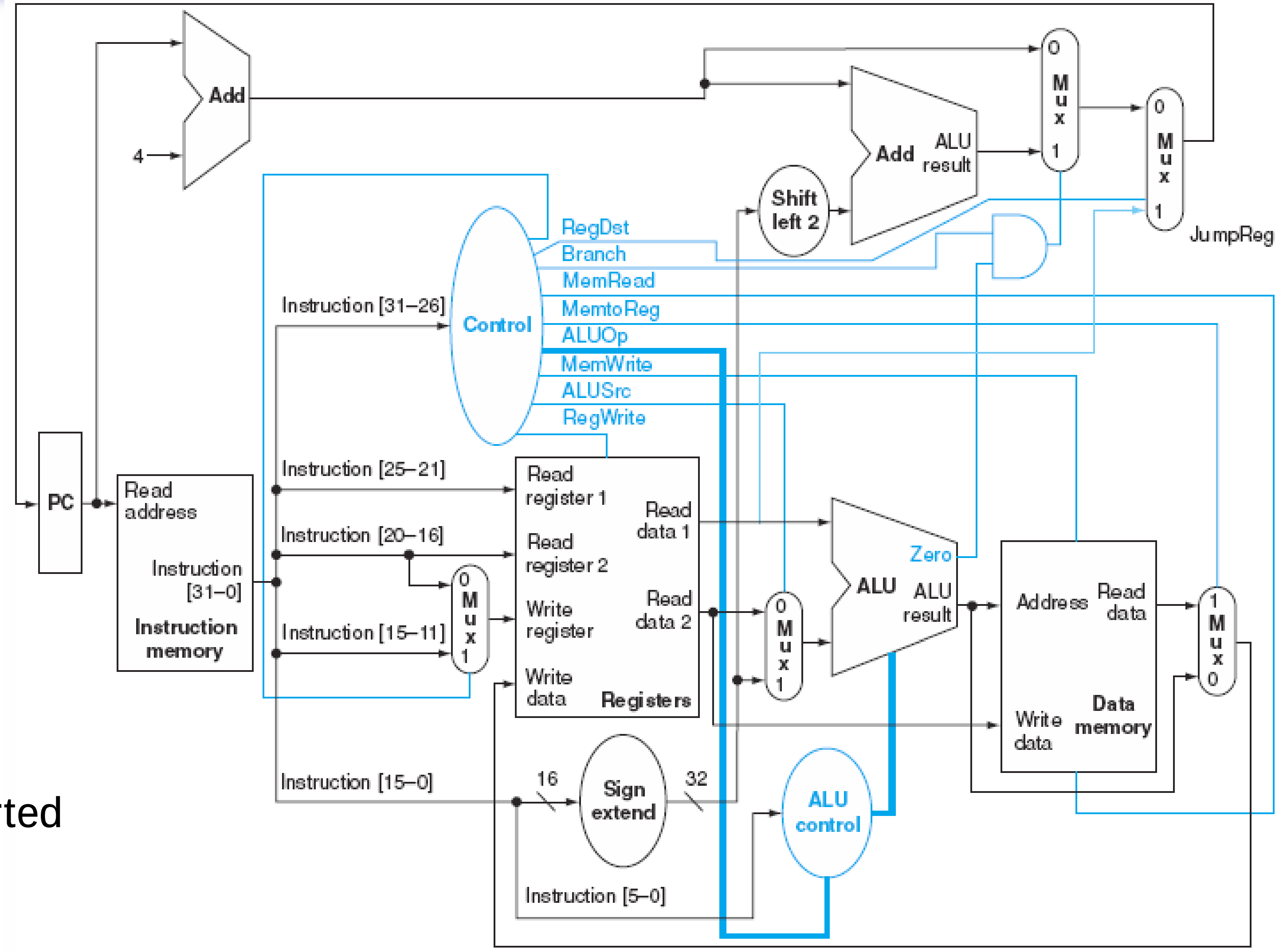
CPI = 20% x 5 + 10% x 4 + 50% x 4 + 30% x 3 = 4 cycles per instruction
ExecTime2= N x 4 / 900 MHz

Speed-Up= ExecTime1/ExecTime2 = 9/8 = 1.125 faster than M1

Single Cycle Datapath and Control 1



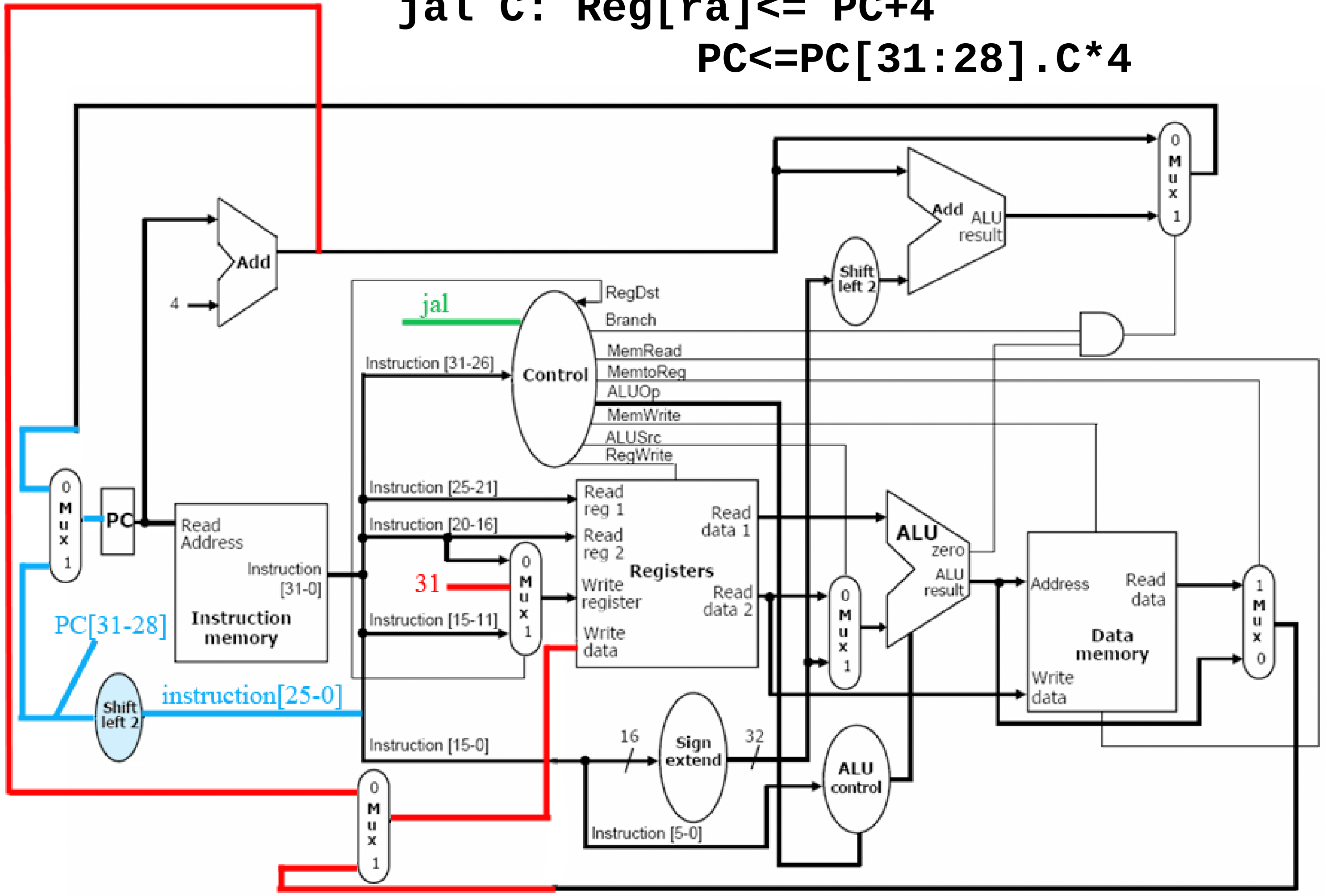
Single Cycle Datapath and Control 2



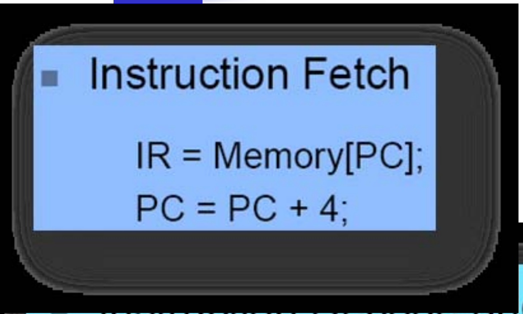
jr inserted

Single Cycle Datapath and Control 3

```
jal C: Reg[ra]<= PC+4
               PC<=PC[31:28].C*4
```

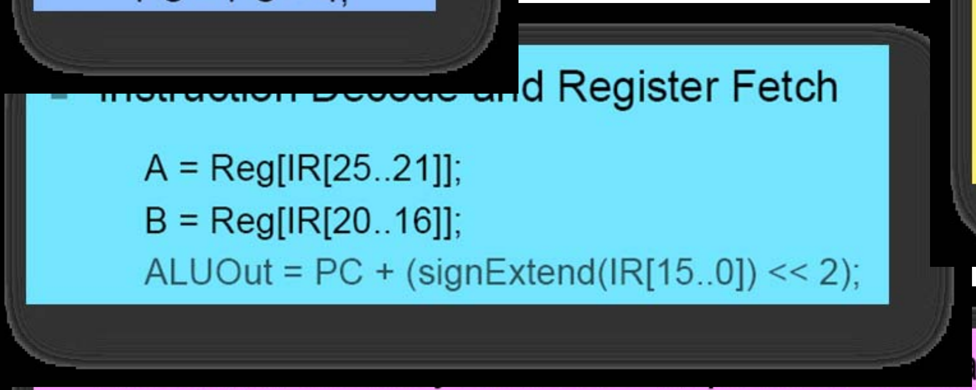


Multi Cycle Datapath and Control 1



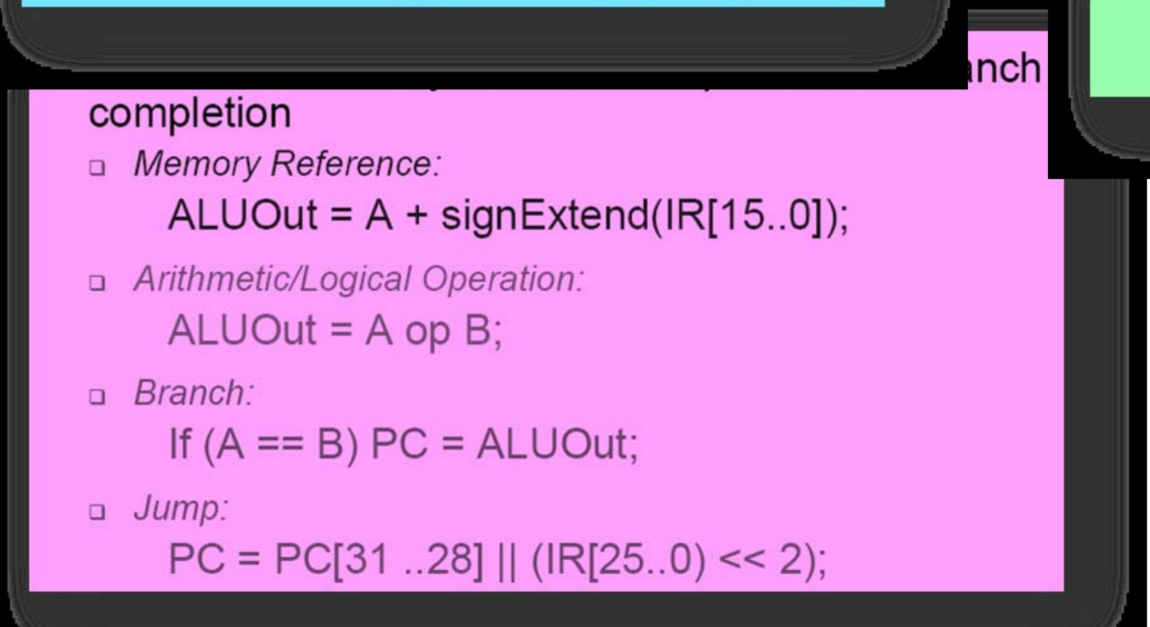
- Instruction Fetch

  IR = Memory[PC];
  PC = PC + 4;



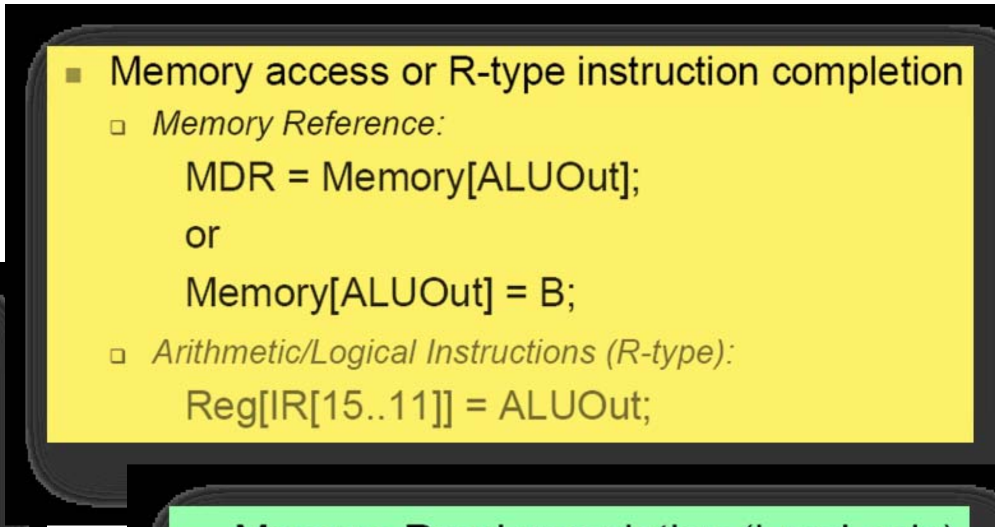
- d Register Fetch

  A = Reg[IR[25..21]];
  B = Reg[IR[20..16]];
  ALUOut = PC + (signExtend(IR[15..0]) << 2);

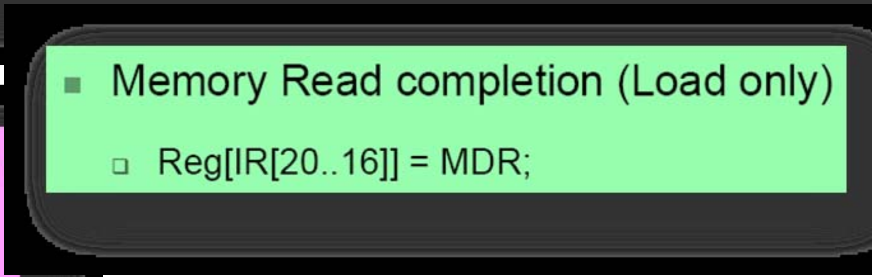


nch completion

- *Memory Reference:*
  ALUOut = A + signExtend(IR[15..0]);
- *Arithmetic/Logical Operation:*
  ALUOut = A op B;
- *Branch:*
  If (A == B) PC = ALUOut;
- *Jump:*
  PC = PC[31 ..28] || (IR[25..0) << 2);

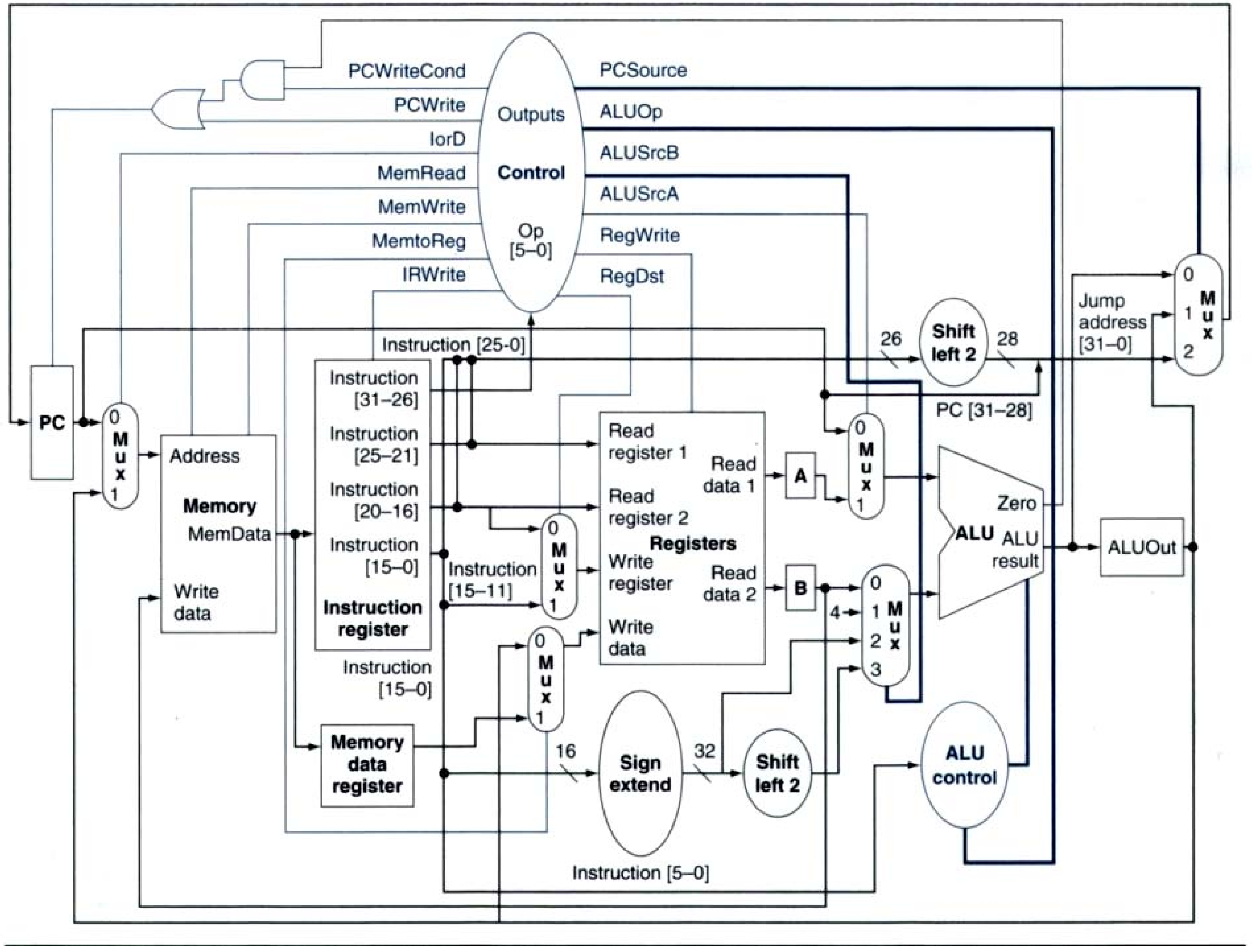


- Memory access or R-type instruction completion
  - *Memory Reference:*
    MDR = Memory[ALUOut];
    or
    Memory[ALUOut] = B;
  - *Arithmetic/Logical Instructions (R-type):*
    Reg[IR[15..11]] = ALUOut;

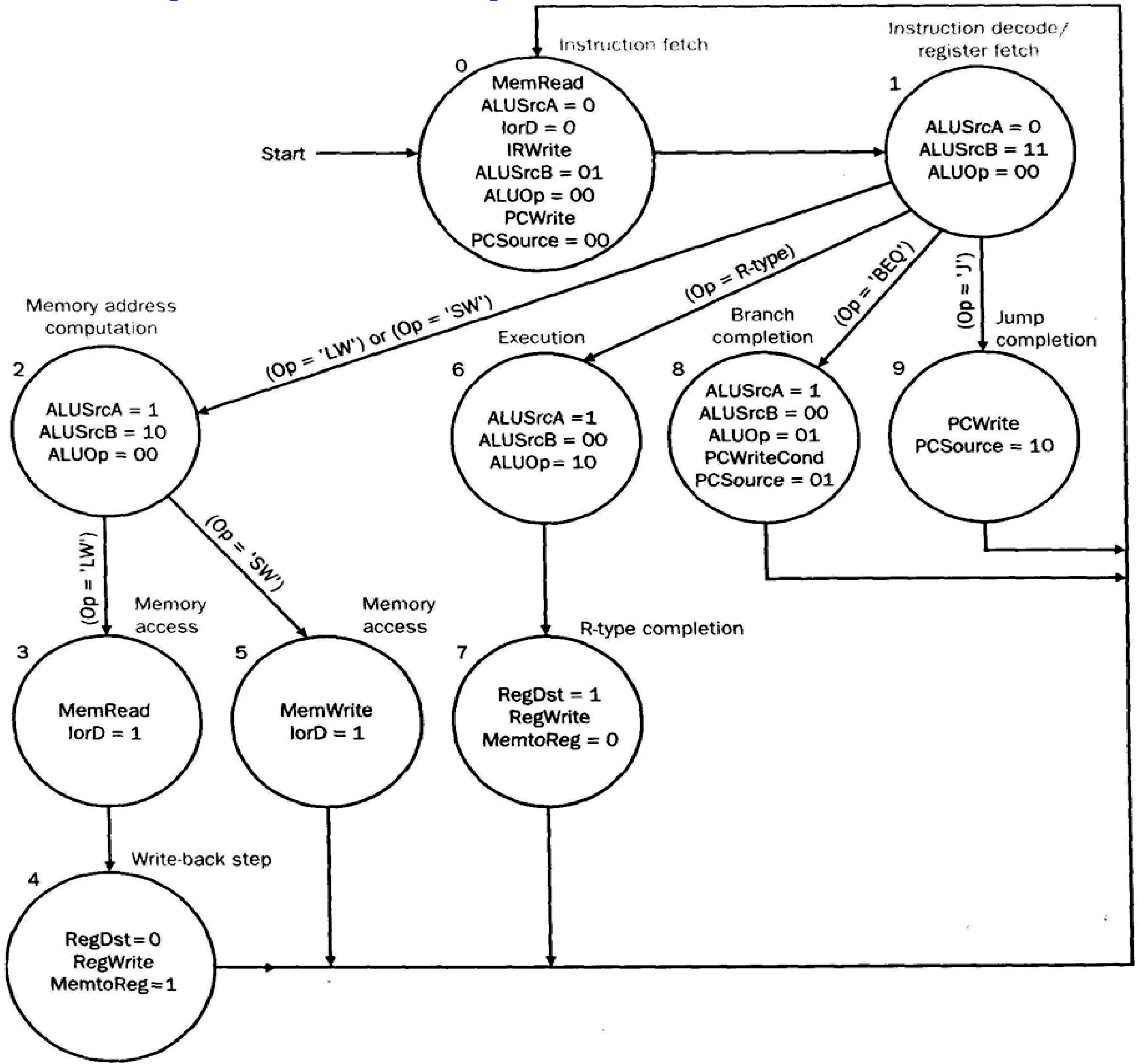


- Memory Read completion (Load only)
  - Reg[IR[20..16]] = MDR;

Multi Cycle Datapath and Control 2



Multi Cycle Datapath and Control 3



Multi Cycle Datapath and Control

```
ldi $rt, 0x12341234: Reg[rt]<=Mem[PC+4]
```

We can use the same datapath.

1. Instruction fetch: Unchanged (IR <= Memory[PC]; PC<= PC + 4).

2. Instruction decode: Unchanged (A <= Reg[IR[25-21]]; B<=REG[IR[20-16]]; ALUOut<=PC+(sign-extend(IR[15-0])<<2).

3. Load immediate value from memory (MDR <= Memory[PC]; PC <= PC + 4).

4. Complete instruction (Reg[IR[20-16]] <= MDR).