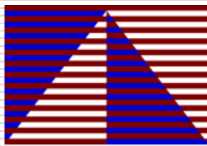
ECE3055
Computer Architecture and Operating Systems

Chapter 5: Datapath Part I



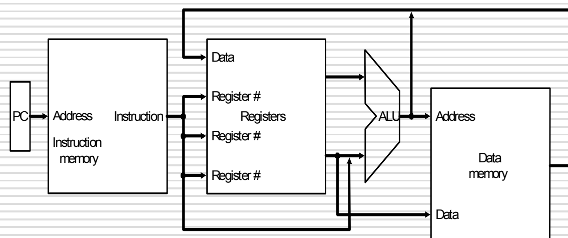
Prof. Hsien-Hsin Sean Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

The Processor: Datapath and Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do
- All instructions use the ALU after reading the registers
  Why? memory-reference? arithmetic? control flow?

More Implementation Details

- Abstract / Simplified View:

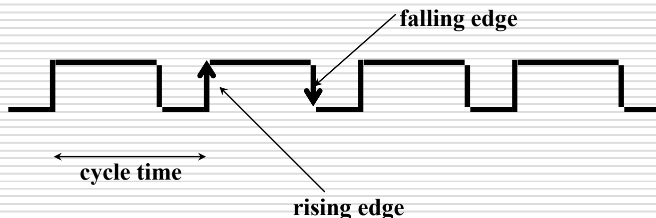


Two types of functional units:

- Elements that operate on data values (combinational)
- Elements that contain state (sequential)

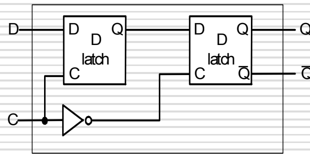
State Elements

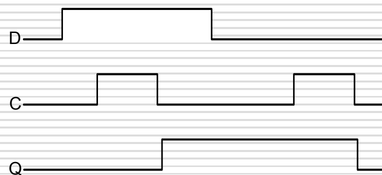
- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



D flip-flop

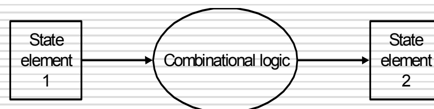
- Output changes only on the clock edge

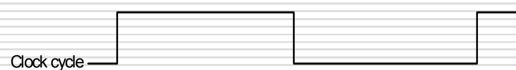




Our Implementation

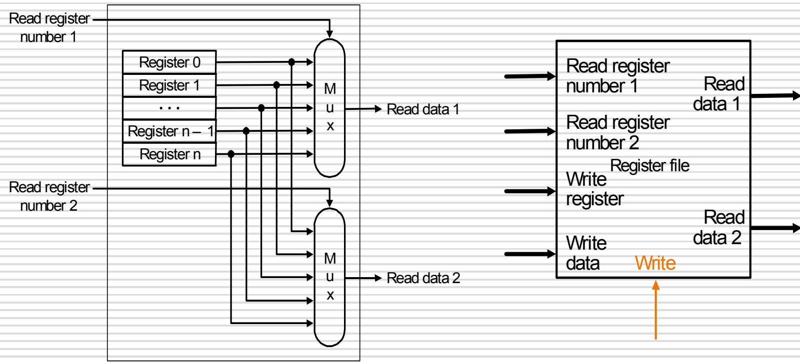
- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements





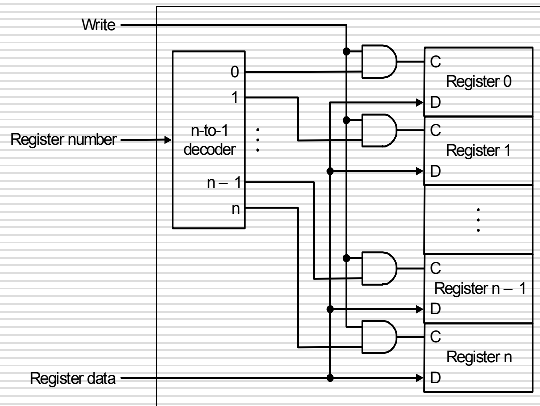
Register File

- Built using D flip-flops (remember ECE 2030!)



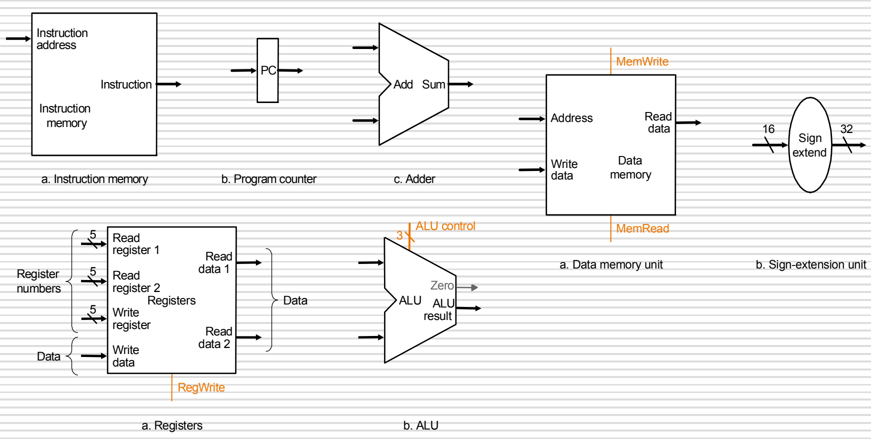
Register File

- Note: we still use the real clock to determine when to write

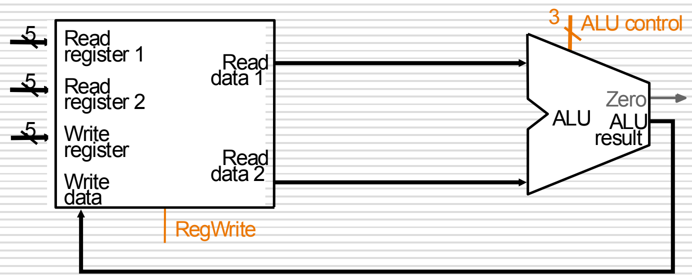


Simple Implementation

- Include the functional units we need for each instruction



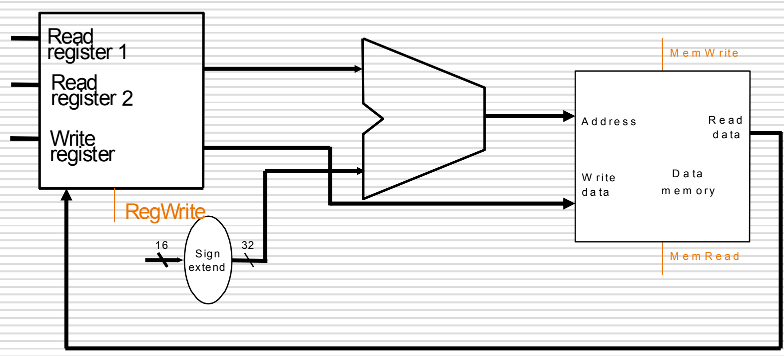
Executing R-Format Instructions

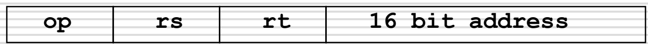


op	rs	rt	rd	shamt	funct

Georgia Institute of Technology

Executing I-Format Instructions

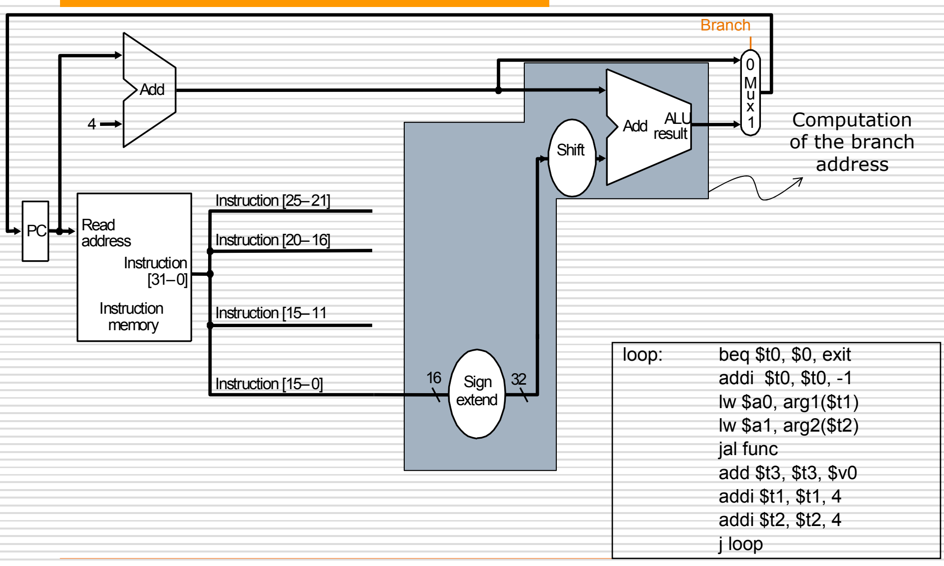




op	rs	rt	16 bit address

Georgia Institute of Technology

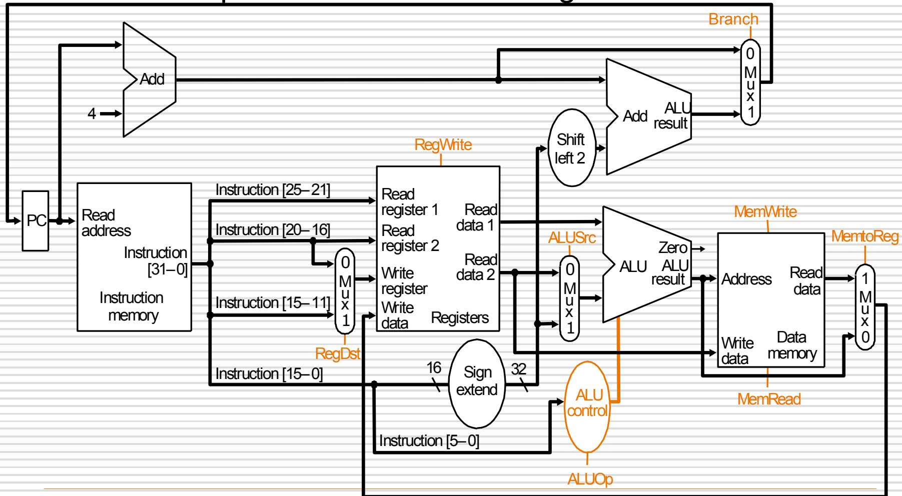
Program Counter Update



Georgia Institute of Technology

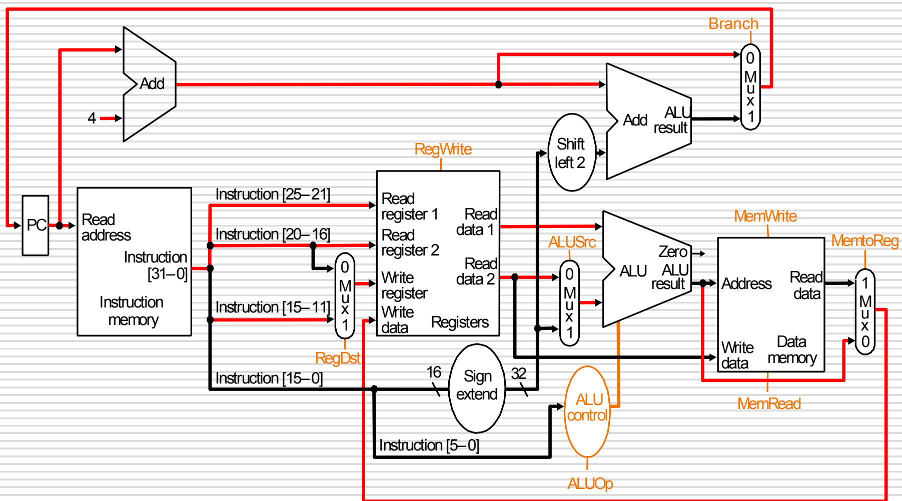
Building the Datapath

- Use multiplexers to stitch them together



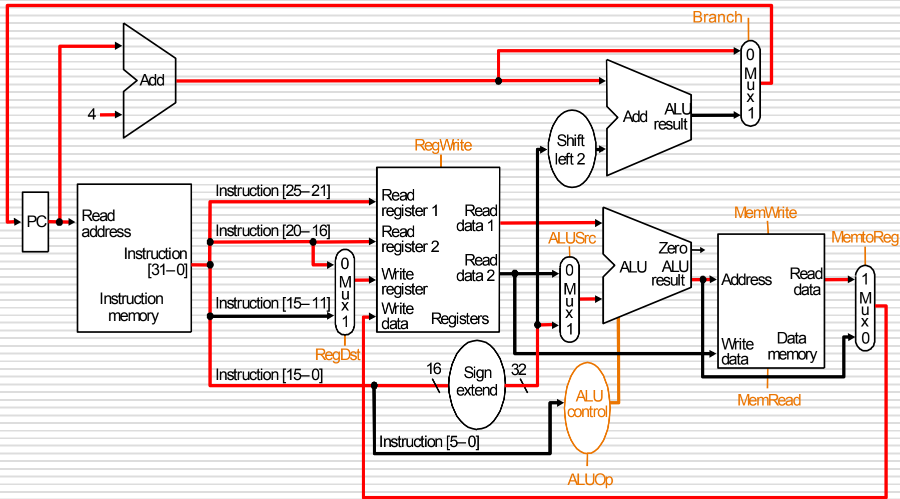
Georgia Institute of Technology

R-Type Instructions
(e.g. add $2, $3, $4; Not JR/JALR)



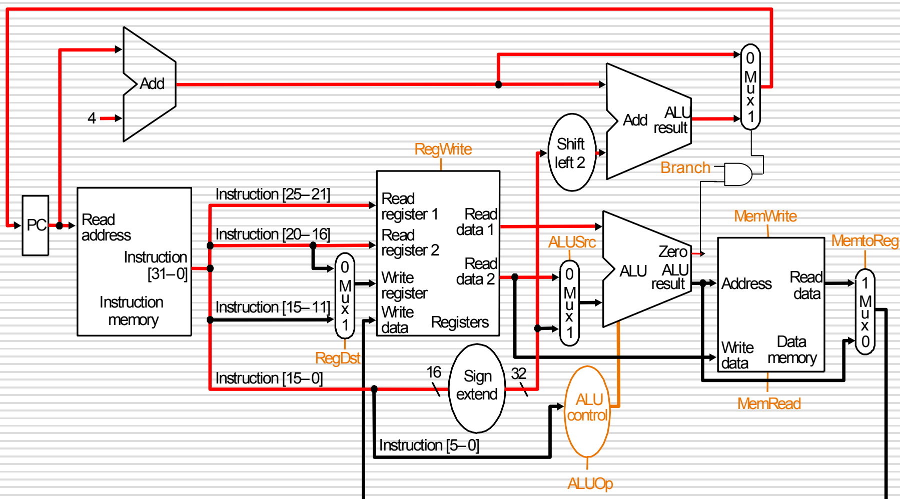
I-Type Instructions

(e.g. lw $4, 1000($15))



I-type Instruction for Branches

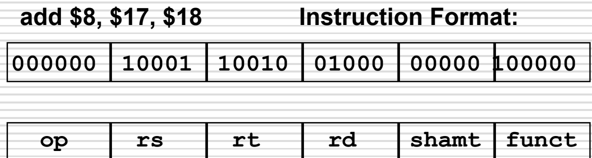
(e.g. beq $4, $5, Label7)



Control

- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexer inputs)
- Information comes from the 32 bits of the instruction
- Example:

add $8, $17, $18 **Instruction Format:**

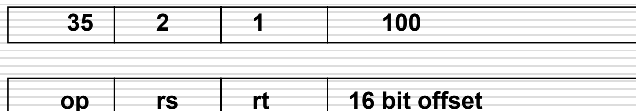


000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU's operation based on instruction type and function code

Control

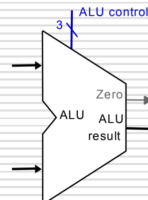
- **e.g., what should the ALU do with this instruction**
- **Example: lw $1, 100($2)**



35	2	1	100
op	rs	rt	16 bit offset

- **ALU control input**

```
000    AND
001    OR
010    add
110    subtract
111    set-on-less-than
```



- **Why is the code for subtract 110 and not 011? What do you need for slt instruction?**

Control the ALU

- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    00 = lw, sw
    01 = beq,
    11 = arithmetic (incl. slt)
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp computed from instruction type

	ALUOp		Funct field						ALU Control	
	ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
lw/sw	0	0	X	X	X	X	X	X	010	add
beq	X	1	X	X	X	X	X	X	110	sub
arith	1	X	X	X	0	0	0	0	010	add
arith	1	X	X	X	0	0	1	0	110	sub
arith	1	X	X	X	0	1	0	0	000	and
arith	1	X	X	X	0	1	0	1	001	or
arith	1	X	X	X	1	0	1	0	111	slt

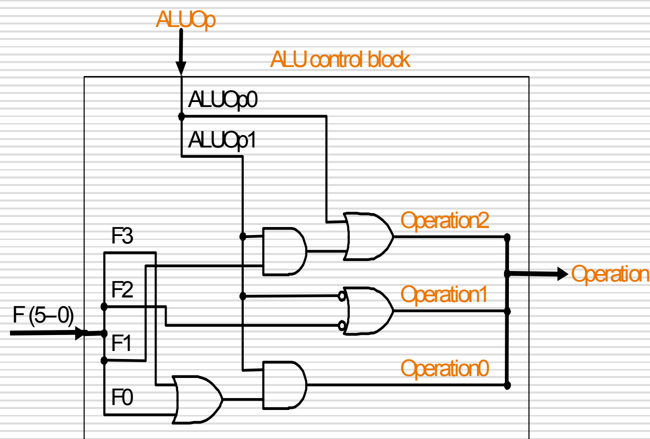
Generated from Decoding inst[31:26] (the ALUOp columns)

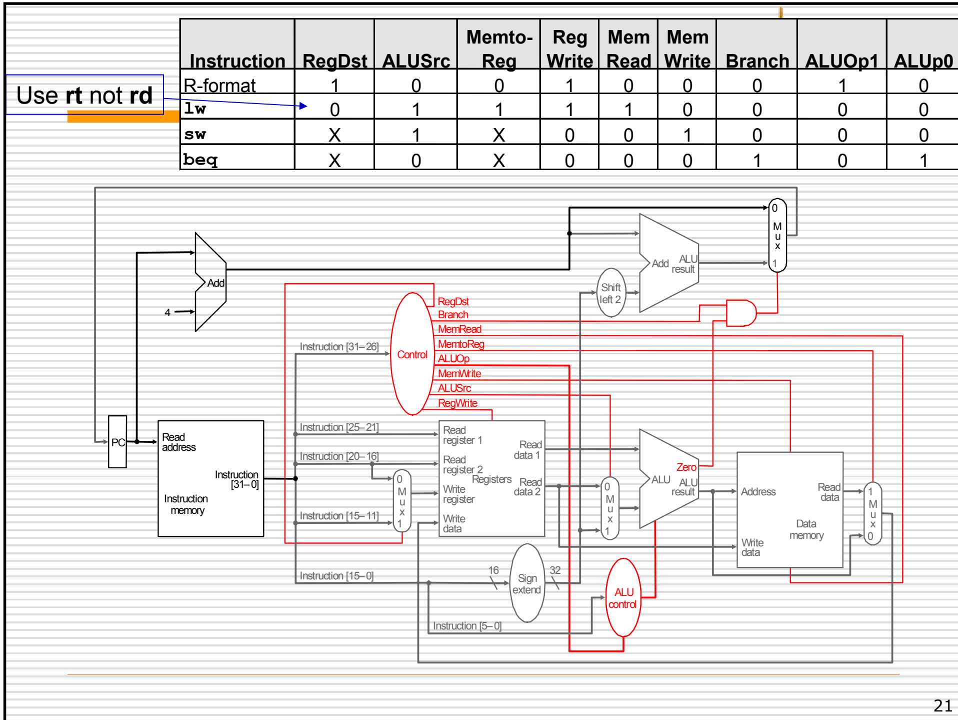
inst[5:0] (the Funct field columns)

ALUOp
ALU control
funct = inst[5:0]
ALU control
3
Zero
ALU
ALU result

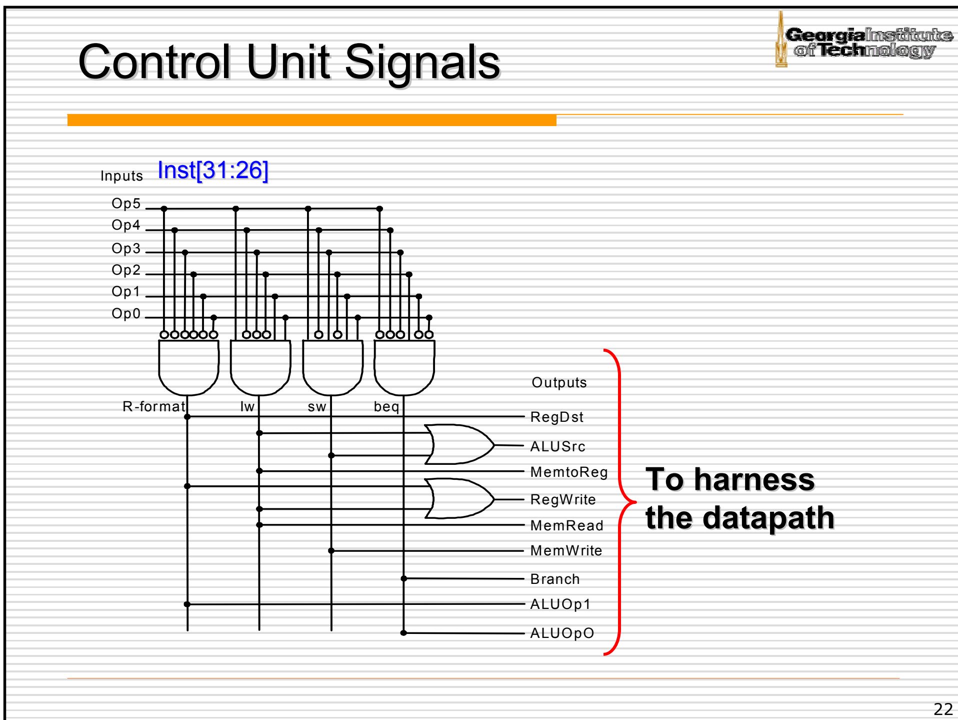
ALU Control

- Simple combinational logic (truth tables)





Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



Controller Implementation

```
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.STD_LOGIC_ARITH.ALL;
USE IEEE.STD_LOGIC_SIGNED.ALL;

ENTITY control IS
  PORT(
      SIGNAL Opcode          : IN     STD_LOGIC_VECTOR( 5 DOWNTO 0 );
      SIGNAL RegDst          : OUT    STD_LOGIC;
      SIGNAL ALUSrc          : OUT    STD_LOGIC;
      SIGNAL MemtoReg        : OUT    STD_LOGIC;
      SIGNAL RegWrite        : OUT    STD_LOGIC;
      SIGNAL MemRead         : OUT    STD_LOGIC;
      SIGNAL MemWrite        : OUT    STD_LOGIC;
      SIGNAL Branch          : OUT    STD_LOGIC;
      SIGNAL ALUop           : OUT    STD_LOGIC_VECTOR( 1 DOWNTO 0 );
      SIGNAL clock, reset    : IN     STD_LOGIC );

END control;
```

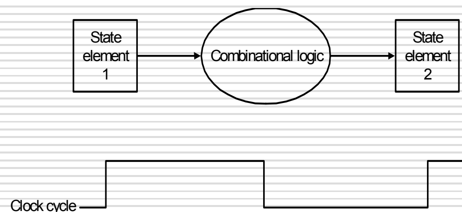

Controller Implementation (cont.)

```
ARCHITECTURE behavior OF control IS

      SIGNAL  R_format, Lw, Sw, Beq    : STD_LOGIC;

BEGIN
                                         -- Code to generate control signals using opcode bits
      R_format   <=  '1'  WHEN  Opcode = "000000"  ELSE '0';
      Lw         <=  '1'  WHEN  Opcode = "100011"  ELSE '0';
      Sw         <=  '1'  WHEN  Opcode = "101011"  ELSE '0';
      Beq        <=  '1'  WHEN  Opcode = "000100"  ELSE '0';
      RegDst     <=  R_format;
      ALUSrc     <=  Lw OR Sw;
      MemtoReg   <=  Lw;
      RegWrite   <=  R_format OR Lw;
      MemRead    <=  Lw;
      MemWrite   <=  Sw;
      Branch     <=  Beq;
      ALUOp( 1 ) <=  R_format;
      ALUOp( 0 ) <=  Beq;

  END behavior;
```

Implementation of each table column

Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



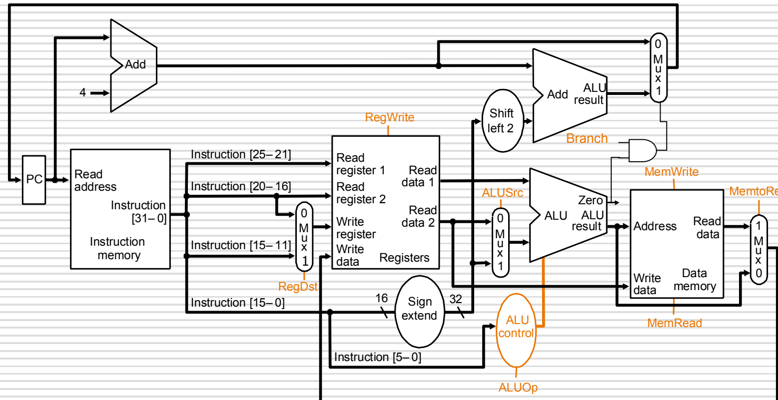
We are ignoring some details like setup and hold times

Extending the Datapath

- Add new instructions
  - j or jal
- Modifications to the datapath
  - Determine data flows
  - Add control elements (muxes)
  - Modify datapath control

Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)



Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a “smaller” cycle time
  - have different instructions take different numbers of cycles
  - a “multicycle” datapath: