Designing MIPS Processor (Single-Cycle)

Dr. Arjan Durresi
Louisiana State University
Baton Rouge, LA 70810
Durresi@Csc.LSU.Edu

These slides are available at:
http://www.csc.lsu.edu/~durresi/CSC3501_07/

Overview

- Datapath
- Control Unit
- Problems with single cycle Datapath

Performance

- Instruction Count
- Clock cycle time
- Clock cycle per Instruction

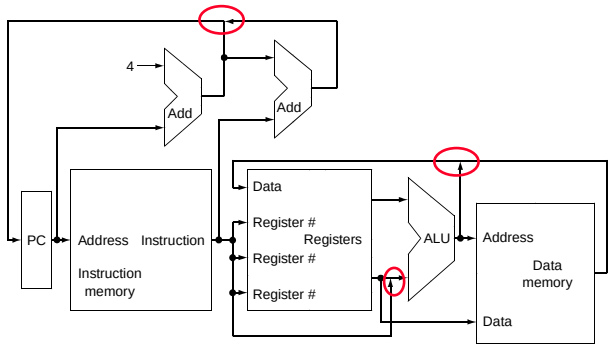
Implementation of Processor

The Processor: Datapath & 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?

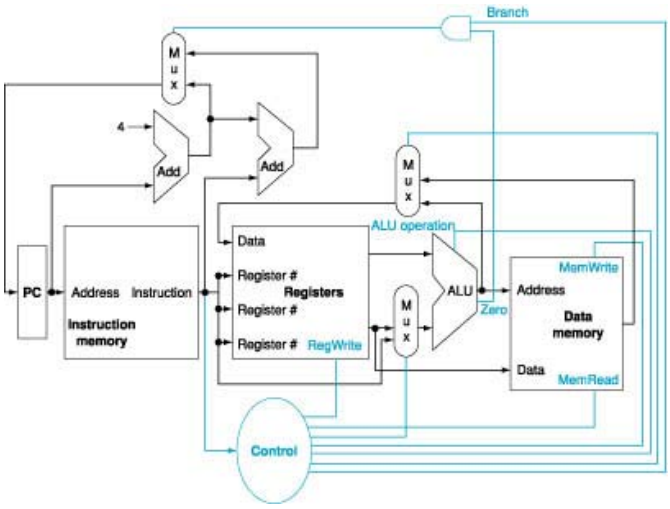
More Implementation Details

- Abstract / Simplified View:



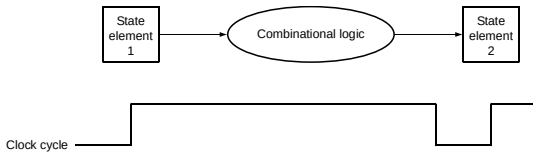
- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

Building the Datapath



Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements at the beginning of the clock cycle,
  - send values through some combinational logic,
  - write results to one or more state elements at the end of the clock cycle.



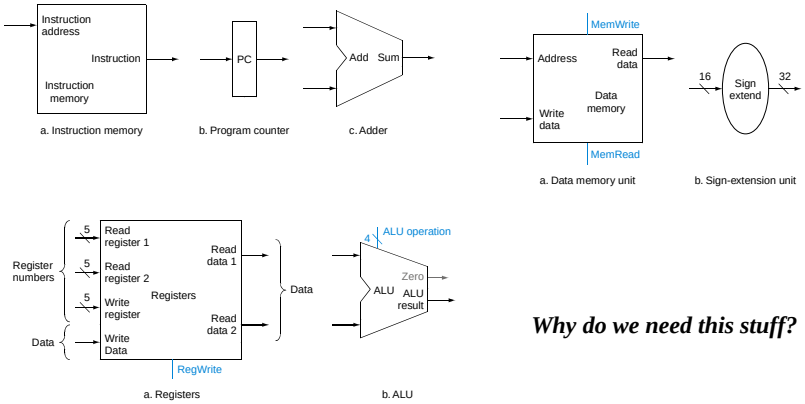
- An edge triggered methodology allows a state element to be read and written in the same clock cycle without creating a race that could to indeterminate data.

Single Cycle Design

- We shall first design a simpler processor that executes each instruction in only one clock cycle time.
- This is not efficient from performance point of view, since:
  - a clock cycle time (i.e. clock rate) must be chosen such that the longest instruction can be executed in one clock cycle
  - makes shorter instructions execute in one unnecessary long cycle.
- Additionally, no resource in the design may be used more than once per instruction, thus some resources will be duplicated.
- Because of that, the singe cycle design will require:
  - two memories (instruction and data),
  - two additional adders.

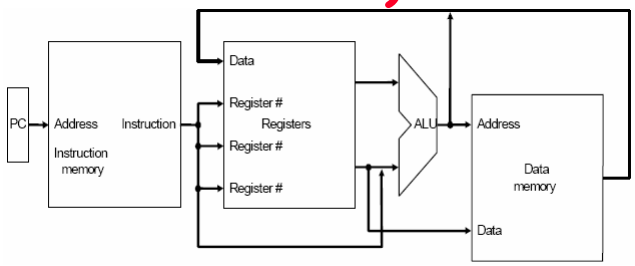
Elements for Datapath Design

- Include the functional units we need for each instruction



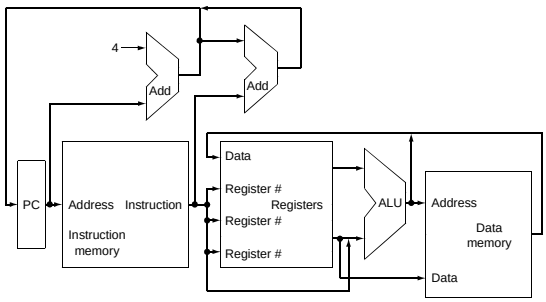
Why do we need this stuff?

Abstract /Simplified View (1st look)



- 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.

Abstract /Simplified View (2nd look)



- PC is incremented by 4, by most instructions, and by 4 + 4×offset, by branch instructions.
- Jump instructions change PC differently (not shown).

Incrementing PC & Fetching Instruction

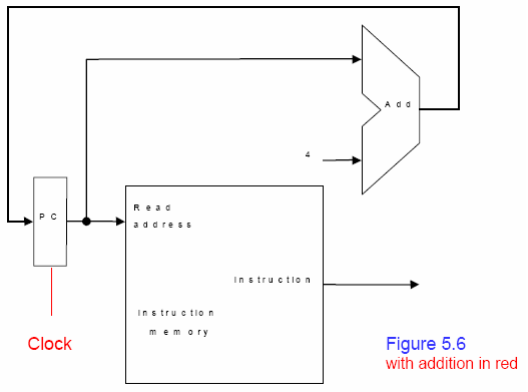
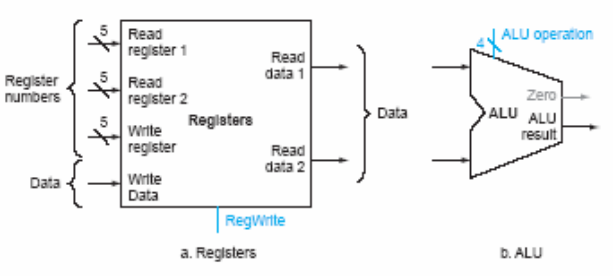


Figure 5.6
with addition in red

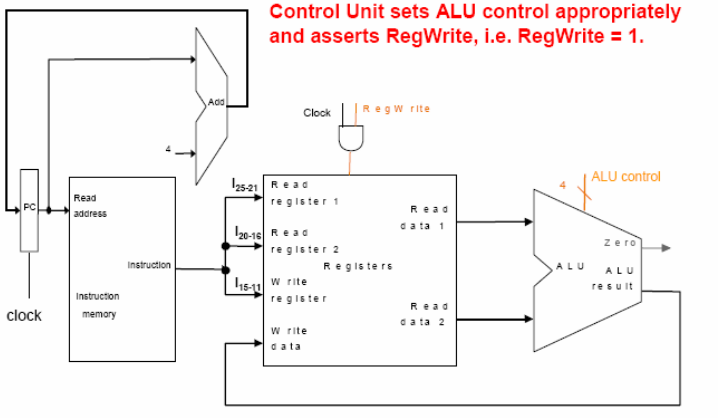
Two elements needed to implement R-formant ALU operations



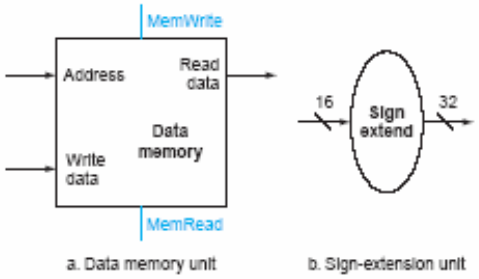

```
add $t1,$t2,$t3
```


Complete Datapath for R-type Instructions

Based on contents of op-code and funct fields, Control Unit sets ALU control appropriately and asserts RegWrite = 1.



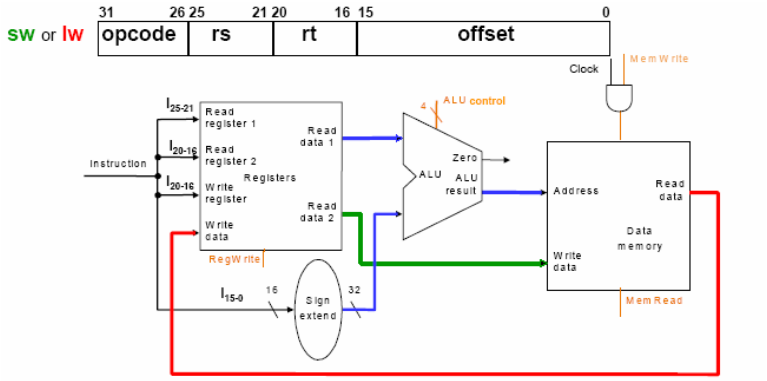
Two elements needed to implement loads and stores




```
lw $t1, offset_value($t2)
Sw $t1, offset_value($t2)
```

Compute a memory address by adding the base register ($t2) to the 16-bit signed offset field contained in the instruction

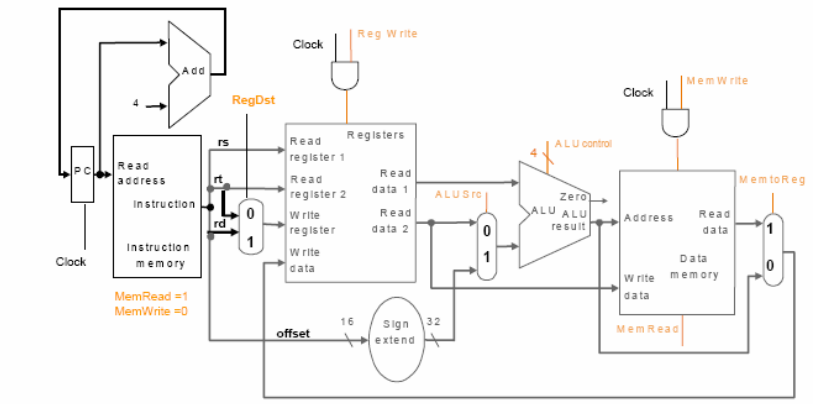
Datapath for LW and SW Instructions



Control Unit sets:

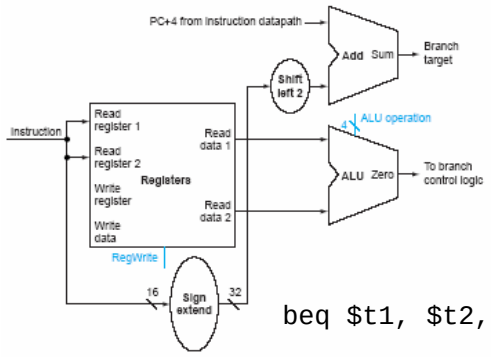
- **ALU control = 0010 (add) for address calculation for both lw and sw**
- **MemRead=0, MemWrite=1 and RegWrite=0 for sw**
- **MemRead=1, MemWrite=0 and RegWrite=1 for lw**

Datapath for R-type, LW & SW Instructions



Let us determine setting of control lines for R-type, lw & sw instructions.

The datapath for a branch




```
beq $t1, $t2, offset
```

Compute the branch target address by adding the sign-extended offset of the instruction to PC

Datapath for R-type, LW, SW & BEQ

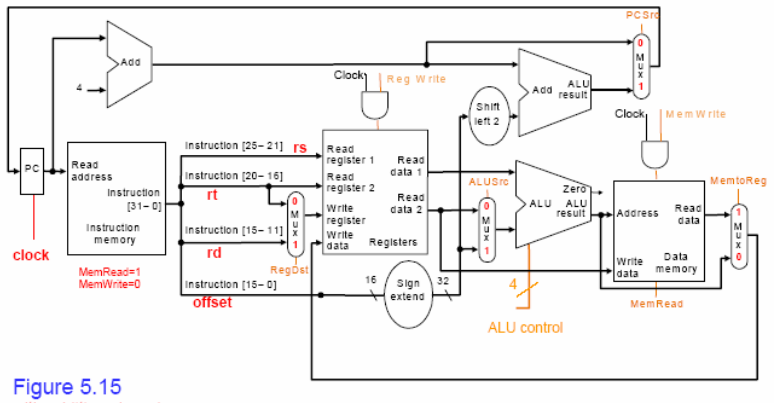
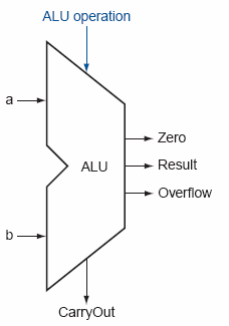


Figure 5.15
with additions in red

ALU

ALU control lines	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	set on less than
1100	NOR

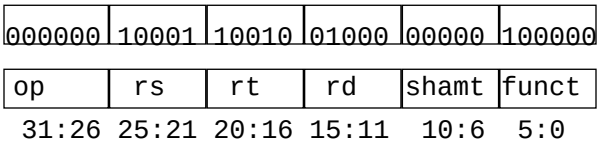


Generate the 4-bit ALU control input using a small control unit that has as inputs the function field of the instruction and a 2-bit control field ALUOp

Control

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

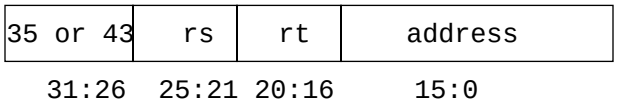
add $8, $17, $18 Instruction Format:



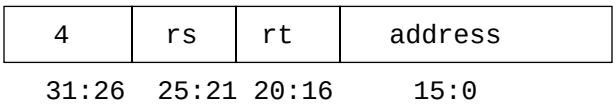
- ALU's operation based on instruction type and function code

Load, store and branch instructions

Load or store instruction



Branch instruction



Control

- Must describe hardware to compute 4-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic

    → **ALUOp computed from instruction type**
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 5.13 The truth table for the three ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Also, when the function field is used, the first two bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Truth Table for (Main) Control Unit

- ALUOp[1-0] = 00 → signal to ALU Control unit for ALU to perform add function, i.e. set Ainvert = 0, Binvert=0 and Operation=10
- ALUOp[1-0] = 01 → signal to ALU Control unit for ALU to perform subtract function, i.e. set Ainvert = 0, Binvert=1 and Operation=10
- ALUOp[1-0] = 10 → signal to ALU Control unit to look at bits $I_{[5-0]}$ and based on its pattern to set Ainvert, Binvert and Operation so that ALU performs appropriate function, i.e. add, sub, slt, and, or & nor

	Input	Output								
	Op-code	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-type	000000	1	0	0	1	**d**	0	0	1	0
lw	100011	0	1	1	1	1	0	0	0	0
sw	101011	d	1	d	0	0	1	0	0	0
beq	000100	d	0	d	0	**d**	0	1	0	1

Truth Table of ALU Control Unit

Input								Output	
ALUOp		Funct field						ALU	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	Control	
0	0	d	d	d	d	d	d	0 0 10	add
0	1	d	d	d	d	d	d	0 1 10	sub
1	0	1	0	0	0	0	0	0 0 10	add
1	0	1	0	0	0	1	0	0 1 10	sub
1	0	1	0	0	1	0	0	0 0 00	and
1	0	1	0	0	1	0	1	0 0 01	or
1	0	1	0	1	0	1	0	0 1 11	slt
1	0	1	0	1	0	0	1	1 1 00	nor

Ainvert Bivert Operation

Design of (Main) Control Unit

Op-code bits 5 4 3 2 1 0	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
0 0 0 0 0 0	1	0	0	1	d	0	0	1	0
1 0 0 0 1 1	0	1	1	1	1	0	0	0	0
1 0 1 0 1 1	d 0	1	d	0	0	1	0	0	0
0 0 0 1 0 0	d 0	0	d	0	d	0	1	0	1
	... 0	... 0							

$$RegDst = \overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,\overline{Op_1}\,\overline{Op_0}$$

$$ALUSrc = Op_5\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}Op_1Op_0 + Op_5\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}Op_1Op_0$$

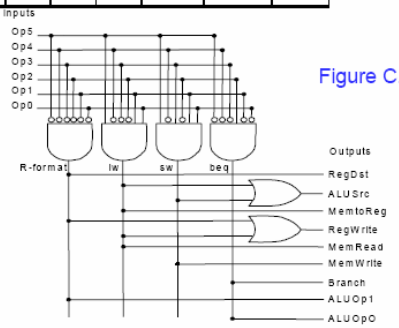
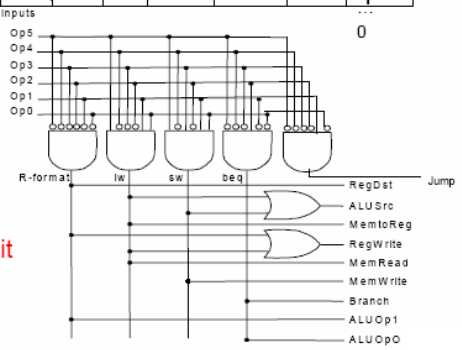


Figure C.2.5

Design of Control Unit (J included)

Op-code bits 5 4 3 2 1 0	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0	Jump
0 0 0 0 0 0	1	0	0	1	d	0	0	1	0	0
1 0 0 0 1 1	0	1	1	1	1	0	0	0	0	0
1 0 1 0 1 1	d	1	d	0	0	1	0	0	0	0
0 0 0 1 0 0	d	0	d	0	d	0	1	0	1	0
J 0 0 0 0 1 0	d	d	d	0	d	0	d	d	d	1

$$Jump = \overline{Op_5}\,\overline{Op_4}\,\overline{Op_3}\,\overline{Op_2}\,Op_1\,\overline{Op_0}$$



No changes in ALU Control unit

Design of 7-Function ALU Control Unit

Input: ALUOp		Input: Funct field						Output: ALU Control	
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0		
0	0	d	d	d	d	d	d	010	add
0	1	d	d	d	d	d	d	110	sub
1	0	1	0	0	0	0	0	010	add
1	0	1	0	0	0	1	0	110	sub
1	0	1	0	0	1	0	0	000	and
1	0	1	0	0	1	0	1	001	or
1	0	1	0	1	0	1	0	111	slt

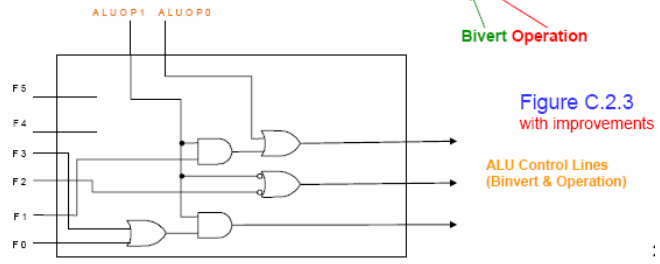


Figure C.2.3
with improvements

ALU Control Lines
(Binvert & Operation)

Cycle Time Calculation

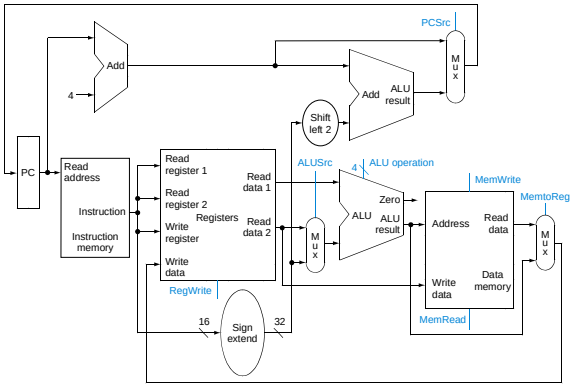
- Let us assume that the **only** delays introduced are by the following tasks:
  - Memory access (read and write time = 3 nsec)
  - Register file access (read and write time = 1 nsec)
  - ALU to perform function (= 2 nsec)
- Under those assumption here are instruction execution times:

	Instr fetch		Reg read		ALU oper		Data memory		Reg write	Total
R-type	3	+	1	+	2	+			1	= 7 nsec
lw	3	+	1	+	2	+	3	+	1	= 10 nsec
sw	3	+	1	+	2	+	3			= 9 nsec
branch	3	+	1	+	2					= 6 nsec
jump	3									= 3 nsec

- Thus a clock cycle time has to be 10nsec, and clock rate = 1/10 nsec = 100MHz

Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (200ps), ALU and adders (100ps), register file access (50ps)



Performance of Single cycle machines

- Memory (200ps), ALU and adders (100ps), register file access (50ps)
- Instructions mix: 25% loads, 10% stores, 45% ALU, 15% branches, 5% jumps.
- Which of the following implementations would be faster?
  - Every instruction operates in a 1 clock cycle of a fixed length
  - Every instruction operates in a 1 clock cycle of a variable length
- CPU execution time = Instruction count x CPI x Clock cycle time
- Since CPI = 1
- CPU execution time = Instruction count x Clock cycle time
- Using the critical paths we can compute the required length for each class:
- R-type 400ps, Load word 600ps, Store word 550ps, Branch 350ps, jump 200ps
- In case 1 the clock has to be 600ps depending on the longest instruction
- A machine with a variable clock will have a clock cycle that varies between 200ps and 600ps.
- The average CPU clock cycle= 600x25% + 550x10% + 400x45% + 350x15%+200x5% = 447.5ps
- So the variable clock machine is faster 1.34 times

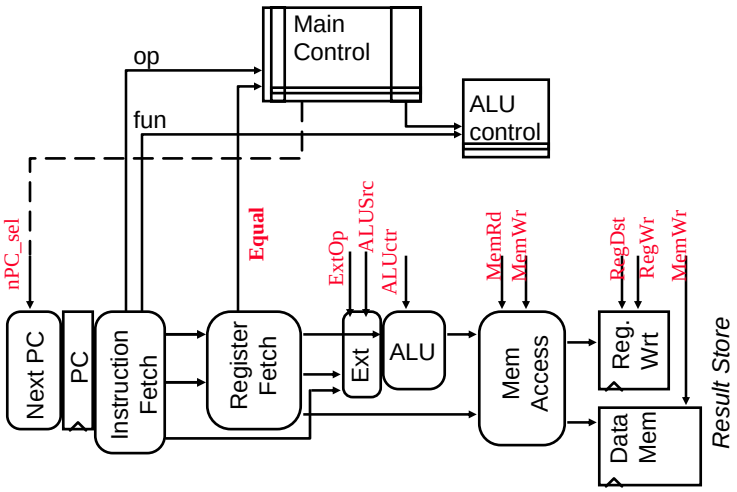
Example

Instruction class	Functional units used by the instruction class				
R-type	Instruction fetch	Register access	ALU	Register access	
Load word	Instruction fetch	Register access	ALU	Register access	Register access
Store word	Instruction fetch	Register access	ALU	Register access	
Branch	Instruction fetch	Register access	ALU		
Jump	Instruction fetch				

Example

Instruction class	Instruction memory	Register read	ALU	Data memory	Register write	Total
R-type	200	50	100	0	50	400ps
Load word	200	50	100	200	50	600ps
Store word	200	50	100	200	0	550ps
Branch	200	50	100	0		350ps
Jump	200					200ps

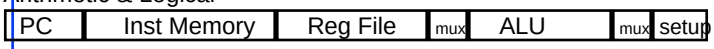
Abstract View of our single cycle process



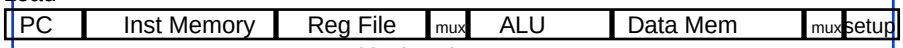
- looks like a FSM with PC as state

What's wrong with our CPI=1 processor?

Arithmetic & Logical

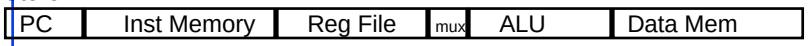


Load

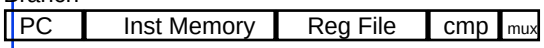


Critical Path

Store



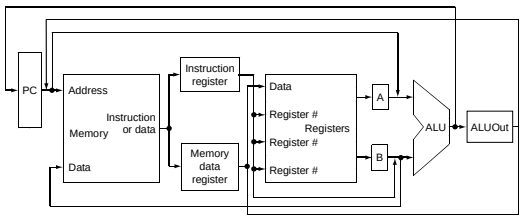
Branch



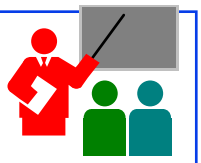
- Long Cycle Time
- All instructions take as much time as the slowest
- Real memory is not as nice as our idealized memory
  - cannot always get the job done in one (short) cycle

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:



Single Cycle Processor: Conclusion



- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - a clock cycle would be much longer,
  - thus for shorter and more often used instructions, such as add & lw, wasteful of time.
- One Solution:
  - use a "smaller" cycle time, and
  - have different instructions take different numbers of cycles.
- And that is a "multi-cycle" processor.