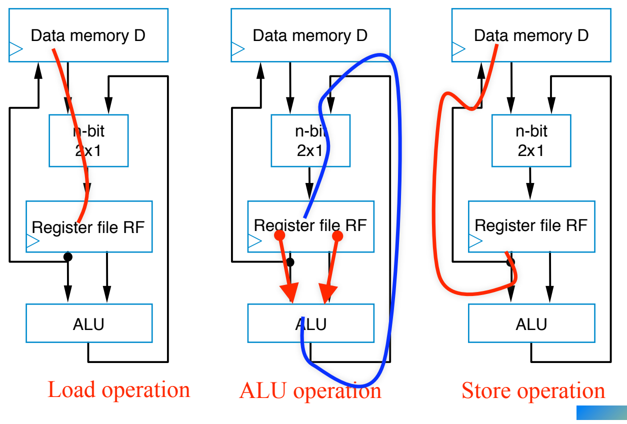
Lecture 15
Midterm Review

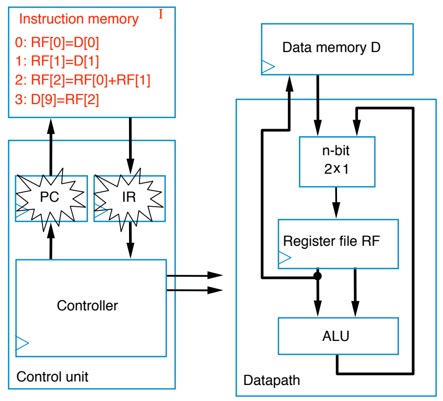
Vahid: Basic Datapath Operations

- Load operation: Load data from data memory to RF
- ALU operation: Transforms data by passing one or two RF register values through ALU, performing operation (ADD, SUB, AND, OR, etc.), and writing back into RF.
- Store operation: Stores RF register value back into data memory
- Each operation can be done in one clock cycle



Vahid: Basic Architecture – Control Unit

- D[9] = D[0] + D[1] – requires a sequence of four datapath operations:
  - 0: RF[0] = D[0]
  - 1: RF[1] = D[1]
  - 2: RF[2] = RF[0] + RF[1]
  - 3: D[9] = RF[2]
- Each operation is an *instruction*
  - Sequence of instructions – *program*
  - Looks cumbersome, but that's the world of programmable processors – Decomposing desired computations into processor-supported operations
  - Store program in *Instruction memory*
  - *Control unit* reads each instruction and executes it on the datapath
    - PC: Program counter – address of current instruction
    - IR: Instruction register – current instruction



Digression:
HW vs. SW based approaches

Foreshadowing:
What if we want ALU to add, subtract? How do we tell it what to do?

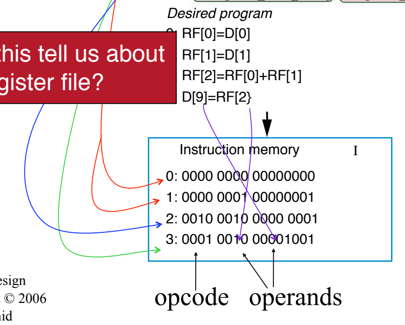
Review: Three-Instruction Programmable Processor

- Instruction Set – List of allowable instructions and their representation in memory, e.g.,
  - *Load* instruction—$0000\ r_3r_2r_1r_0\ d_7d_6d_5d_4d_3d_2d_1d_0$
  - *Store* instruction—$0001\ r_3r_2r_1r_0\ d_7d_6d_5d_4d_3d_2d_1d_0$
  - *Add* instruction—$0010\ ra_3ra_2ra_1ra_0\ rb_3rb_2rb_1rb_0\ rc_3rc_2rc_1rc_0$

What does this tell you about data memory?

What does this tell us about the register file?

"Instruction" is an idea that helps abstract 1s, 0s, but still provides info. about HW



Instructions in 0s and 1s – *machine code*

Review: Assembly Code

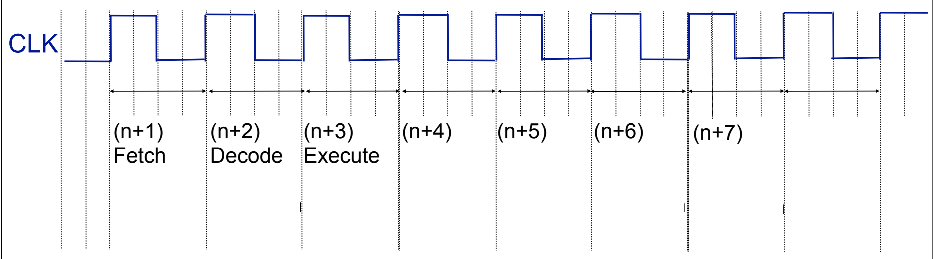
- Machine code (0s and 1s) hard to work with
- Assembly code – Uses mnemonics
  - *Load* instruction—**MOV Ra, d**
    - specifies the operation *RF[a]=D[d]*. *a* must be 0,1, ..., or 15—so *R0* means *RF[0]*, *R1* means *RF[1]*, etc. *d* must be 0, 1, ..., 255
  - • *Store* instruction—**MOV d, Ra**
    - specifies the operation *D[d]=RF[a]*
  - • *Add* instruction—**ADD Ra, Rb, Rc**
    - specifies the operation *RF[a]=RF[b]+RF[c]*

Desired program		
0: RF[0]=D[0]	0: 0000 0000 00000000	0: MOV R0, 0
1: RF[1]=D[1]	1: 0000 0001 00000001	1: MOV R1, 1
2: RF[2]=RF[0]+RF[1]	2: 0010 0010 0000 0001	2: ADD R2, R0, R1
3: D[9]=RF[2]	3: 0001 0010 00001001	3: MOV 9, R2
	machine code	assembly code

Exercise: Understanding the Processor Design (2)

Q1: D[8] = D[8] + RF[1] + RF[4]

…	
I[15]: Add R2, R1, R4	**RF[1] = 4**
I[16]: MOV R3, 8	**RF[4] = 5**
I[17]: Add R2, R2, R3	**D[8] = 7**
…	



A Six-Instruction Programmable Processor

- Let's add three more instructions:
  - *Load-constant* instruction—**0011 $r_3r_2r_1r_0$ $c_7c_6c_5c_4c_3c_2c_1c_0$**
    - **MOV Ra, #c**—specifies the operation *RF[a]=c*
  - *Subtract* instruction—**0100 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**
    - **SUB Ra, Rb, Rc**—specifies the operation *RF[a]=RF[b] – RF[c]*
  - *Jump-if-zero* instruction—**0101 $ra_3ra_2ra_1ra_0$ $o_7o_6o_5o_4o_3o_2o_1o_0$**
    - **JMPZ Ra, offset**—specifies the operation *PC = PC + offset* if *RF[a]* is 0

TABLE 8.1 Six-instruction instruction set..

Instruction	Meaning
MOV Ra, d	RF[a] = D[d]
MOV d, Ra	D[d] = RF[a]
ADD Ra, Rb, Rc	RF[a] = RF[b]+RF[c]
MOV Ra, #C	RF[a] = C
SUB Ra, Rb, Rc	RF[a] = RF[b]-RF[c]
JMPZ Ra, offset	PC=PC+offset if RF[a]=0

TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

Controller FSM for the Six-Instruction Processor

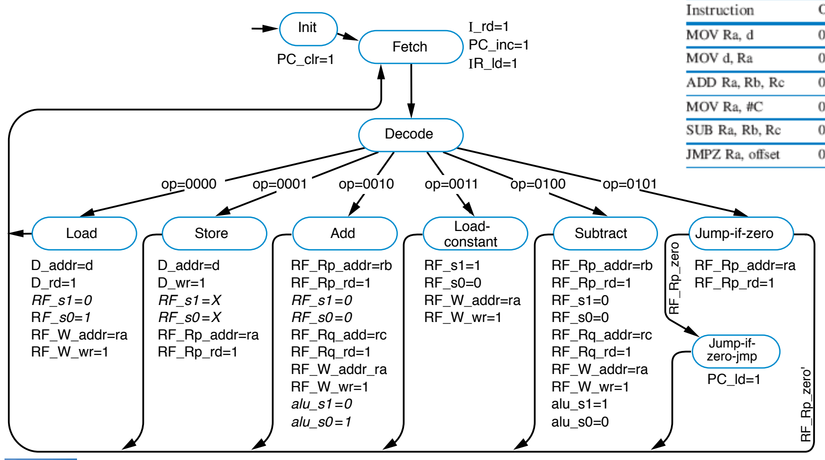


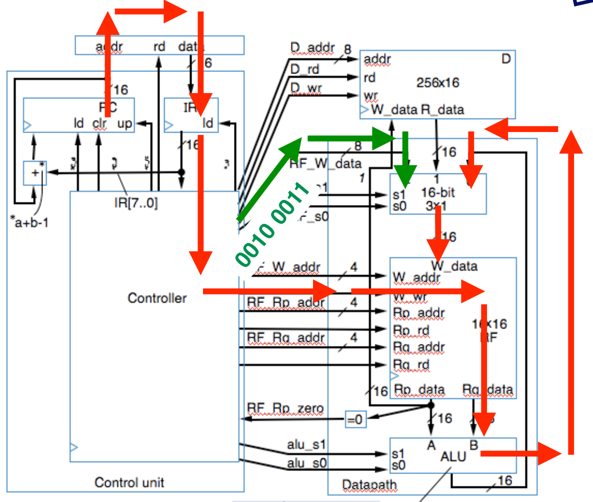
TABLE 8.2 Instruction opcodes.

Instruction	Opcode
MOV Ra, d	0000
MOV d, Ra	0001
ADD Ra, Rb, Rc	0010
MOV Ra, #C	0011
SUB Ra, Rb, Rc	0100
JMPZ Ra, offset	0101

A quick look: more complex ISAs

Datapath

Path of Add from start to finish.



Add: 0010 0001 0010 0011

Bits for Load C also "sent" 0010 0011

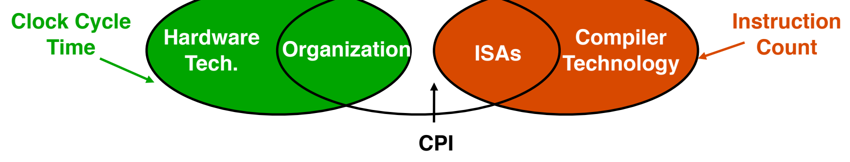
s1	s0	ALU operation
0	0	pass A through
0	1	A+B
1	0	A-B

More types = more multiplexor inputs, signal routing

A CPU : The Bigger Picture

$$\frac{Instructions}{Program} \times \frac{Clock\ cycles}{Instruction} \times \frac{Seconds}{Clock\ Cycle} = \frac{Seconds}{Program} = CPU\ time$$

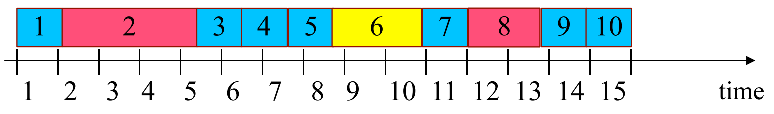
- We can see CPU performance dependent on:
  - Clock rate, CPI, and instruction count
- CPU time is directly proportional to all 3:
  - Therefore an x % improvement in any one variable leads to an x % improvement in CPU performance
- But, everything usually affects everything:



IC, CPI and IPC

Consider the processor we have worked on.

What is its CPI? IPC?



Total Execution Time = 15 cycles

Instruction Count (IC) = Number of Instructions = 10

Average number of cycles per instruction (CPI) =

Instructions per Cycle (IPC) =

Can CPI < 1?

Different Types of Instructions

- Multiplication takes more time than addition
- Floating point operations take longer than integer operations
- Memory accesses take more time than register accesses
- NOTE: changing the cycle time often affects the number of cycles an instruction will take

$$\text{CPU Clock Cycles} = \sum_{i=1}^{n} CPI_i * IC_i = AvgCPI * IC$$

Question 2a - Measurement Comparison

- Given that two machines have the same ISA, which measurement is always the same for both machines running program P?
  - Clock Rate:
  - CPI:
  - Execution Time:
  - Number of Instructions:
  - MIPS:

Deriving the previous formula

$$\text{Speedup}_{\text{overall}} = \frac{\text{Execution Time}_{\text{old}}}{\text{Execution Time}_{\text{new}}} = \boxed{\frac{1}{(1 - \text{Fraction}_{\text{enhanced}}) + \dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}}$$

$$\frac{1}{\underbrace{(1 - \text{Fraction}_{\text{enhanced}})}_{} + \boxed{\dfrac{\text{Fraction}_{\text{enhanced}}}{\text{Speedup}_{\text{enhanced}}}}}$$

1 ← normalized old execution time

1 - % enhanced (i.e. part of the task will take the same amount of time as before)

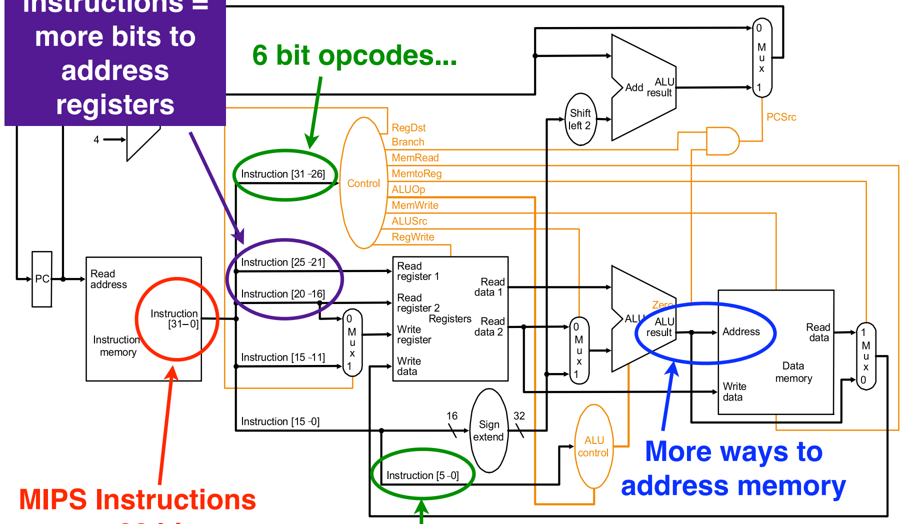
% of task that will run faster / how much faster it will run

(note: # should be < 1)
(otherwise, performance gets worse)
(represents new component of ex. time)

MIPS Datapath

Longer instructions = more bits to address registers

6 bit opcodes...



More ways to address memory

MIPS Instructions are 32 bits

... plus 6 bit function codes = more functionality

MIPS Registers

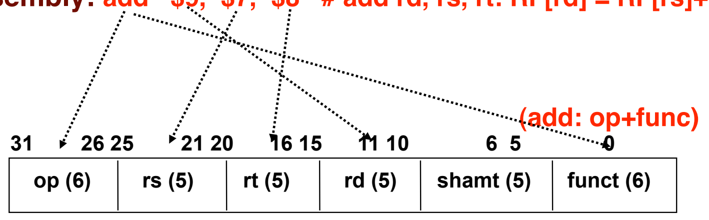
(and the "conventions" associated with them)

Name	R#	Usage	Preserved on Call
$zero	0	The constant value 0	n.a.
$at	1	Reserved for assembler	n.a.
$v0-$v1	2-3	Values for results & expr. eval.	no
$a0-$a3	4-7	Arguments	no
$t0-$t7	8-15	Temporaries	no
$s0-$s7	16-23	Saved	yes
$t8-$t9	24-25	More temporaries	no
$k0-$k1	26-27	Reserved for use by OS	n.a.
$gp	28	Global pointer	yes
$sp	29	Stack pointer	yes
$fp	30	Frame pointer	yes
$ra	31	Return address	yes

R-Type: Assembly and Machine Format

- R-type: All operands are in registers

Assembly: add $9, $7, $8 # add rd, rs, rt: RF[rd] = RF[rs]+RF[rt]



Machine:

B:	000000	00111	01000	01001	xxxxx	100000
D:	0	7	8	9	x	32

R-type Instructions

- All instructions have 3 operands
- All operands must be registers
- Operand order is fixed (destination first)
- Example:

  C code: `A = B - C;`

  (Assume that A, B, C are stored in registers s0, s1, s2.)

  MIPS code: sub $s0, $s1, $s2

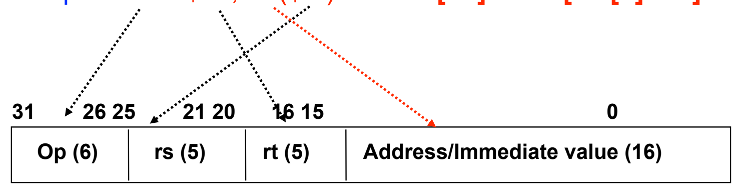
  Machine code: 000000 10001 10010 10000 xxxxx 100010

- Other R-type instructions
  - addu, mult, and, or, sll, srl, …

I-Type Instructions: Another Example

- I-type: One operand is an immediate value and others are in registers

Example: lw $s3, 32($t0) # RF[19] = DM[RF[8]+32]

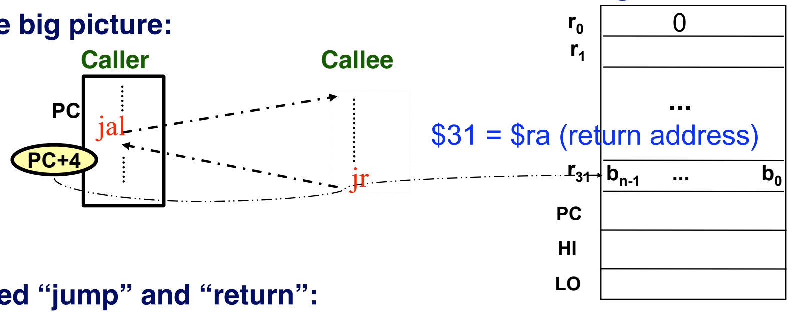


B:	100011	01000	10011	0000000000100000
D:	35	8	19	32

How about load the next word in memory?

MIPS Procedure Handling

- The big picture:



- Need "jump" and "return":
  - *jal ProcAddr* # issued in the caller
    - jumps to ProcAddr
    - save the return instruction address in $31
    - PC = JumpAddr, RF[31]=PC+4;
  - *jr $31* ($ra) # last instruction in the callee
    - jump back to the caller procedure
    - PC = RF[31]

More complex cases

- Register contents across procedure calls are designated as either caller or callee saved
- MIPS register conventions:
  - $t*, $v*, $a*: not preserved across call
    - caller saves them if required
  - $s*, $ra, $fp: preserved across call
    - callee saves them if required
  - See P&H FIGURE 2.18 (p.88) for a detailed register usage convention
  - Save to where??
- More complex procedure calls
  - What if your have more than 4 arguments?
  - What if your procedure requires more registers than available?
  - What about nested procedure calls?
  - What happens to $ra if proc1 calls proc 2 which calls proc3,…

Procedure call essentials: Caller/Callee Mechanics

- **Four places**

Who does what when?

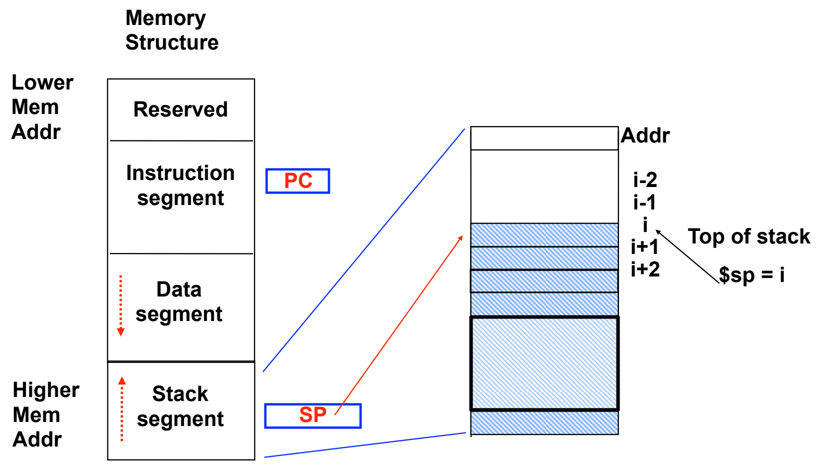
```
foo()
{
    1. caller at call time
    bar(42);
    4. caller after return
}
```

```
bar(int a)
{
    2. callee at entry
    int temp = 3;
    ...
    return(temp + a);
    3. callee at exit
}
```

The stack comes to the rescue

- Stack
  - A dedicated area of memory
  - First-In-Last-Out (FILO)
  - Used to
    - Hold values passed to a procedure as arguments
    - Save register contents when needed
    - Provide space for variables local to a procedure
- Stack operations
  - push: place data on stack (sw in MIPS)
  - pop: remove data from stack (lw in MIPS)
- Stack pointer
  - Stores the address of the top of the stack
  - $29 ($sp) in MIPS

Where is the stack located?



Memory Structure
Lower Mem Addr
Higher Mem Addr
Reserved
Instruction segment
Data segment
Stack segment
PC
SP
Addr
i-2
i-1
i
i+1
i+2
Top of stack
$sp = i