Lecture 15
Midterm Review

Some slides/images from Vahid text – hence this notice:

Stored Programs

A hypothetical translation:

```
for i=0; i<5; i++ {
      a = (a*b) + c;
}
```

```
MULT temp,a,b  # temp ← a*b
MULT r1,r2,r3  # r1 ← r2*r3
```

```
ADD a,temp,c   # a ← temp+c
ADD r2,r1,r4   # r2 ← r1+r4
```

Can define codes for MULT and ADD
Assume MULT = 110011 & ADD = 001110

stored program becomes

PC	110011	000001	000010	000011
PC+1	001110	000010	000001	000100

Program = sequence of instructions

- **Instruction Set – List of allowable inst**
  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?

Desired program
RF[0]=D[0]
RF[1]=D[1]
RF[2]=RF[0]+RF[1]
3: D[9]=RF[2]

Instruction memory I
0: 0000 0000 00000000
1: 0000 0001 00000001
2: 0010 0010 0000 0001
3: 0001 0010 00001001

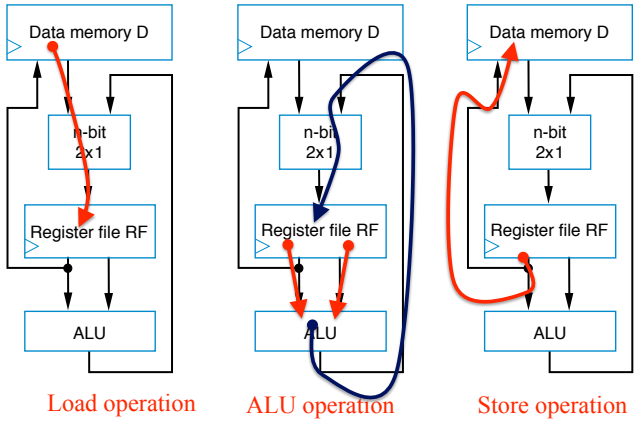
opcode operands

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

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

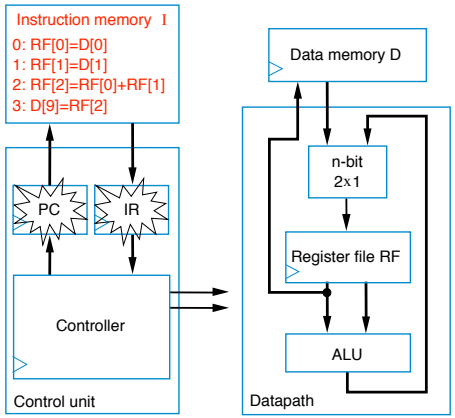
Datapath works on instruction encoding

- **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**



Control logic ensures datapath processes instruction correctly

- **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: Instruc… instructio…**

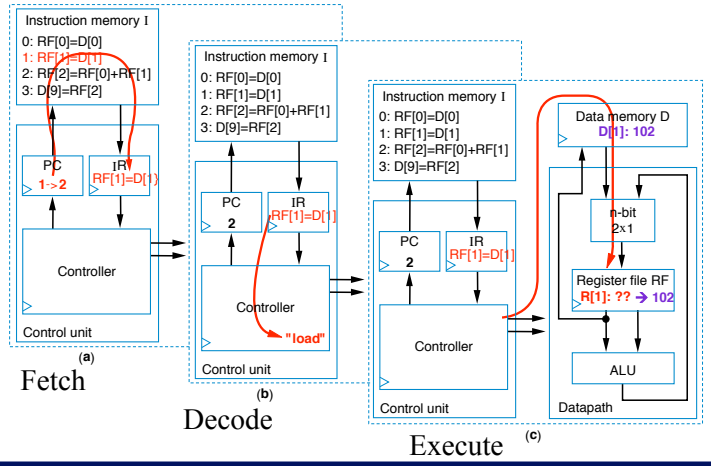


Digression: HW vs. SW based approaches

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

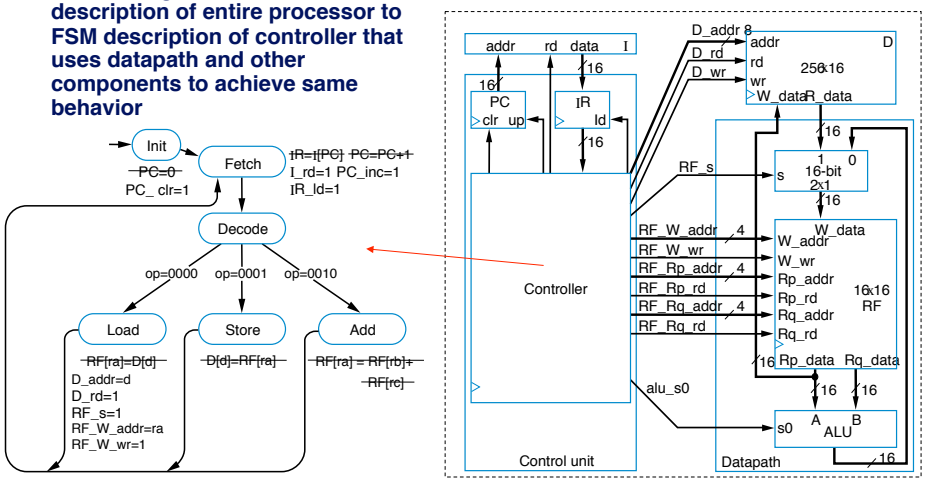
Control signals must arrive at right time

- **To carry out *each instruction*, the control unit must:**
  - **Fetch – Read instruction from inst. mem.**
  - **Decode – Determine the operation and operands of the instruction**
  - **Execute – Carry out the instruction's operation using the datapath**



Control signals must arrive at right time

- **Convert high-level state machine description of entire processor to FSM description of controller that uses datapath and other components to achieve same behavior**

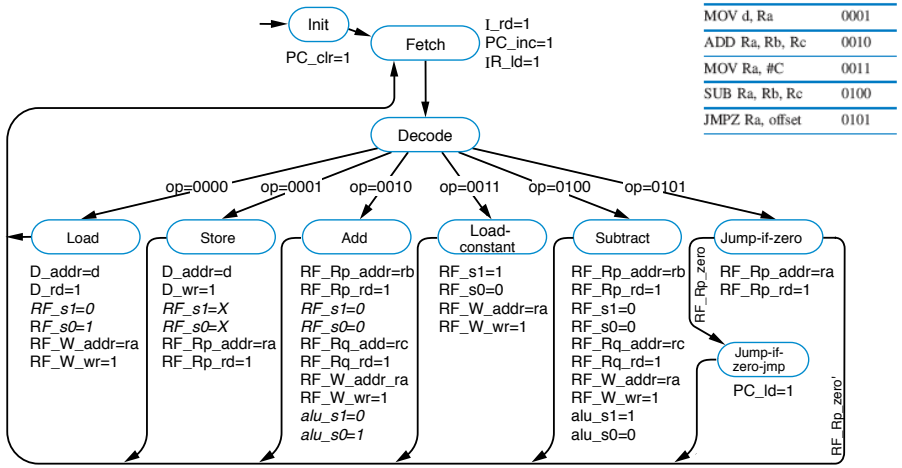


More complex state diagram

State diagram tells you how many CCs instruction takes; what control signals must be generated in each state

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



Generally, register data written in CC N, available in CC N+1

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

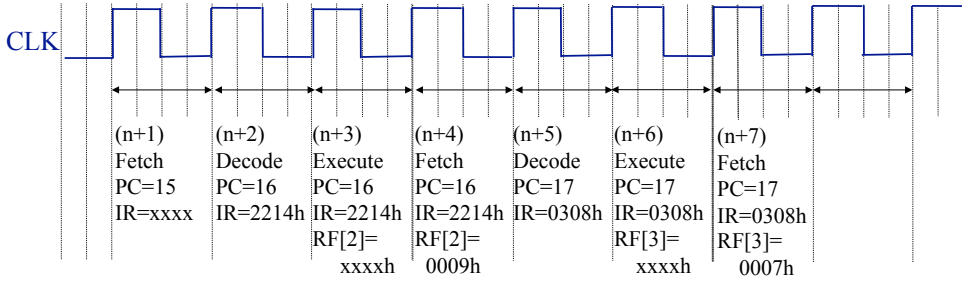
…

I[15]: Add R2, R1, R4 RF[1] = 4

I[16]: MOV R3, 8 RF[4] = 5

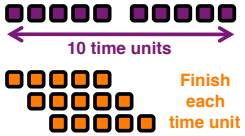
I[17]: Add R2, R2, R3 D[8] = 7

…



Common (and good) performance metrics

- latency: response time, execution time
  - good metric for fixed amount of work (minimize time)
- throughput: bandwidth, work per time, "performance"
  - = (1 / latency) when there is NO OVERLAP
  - > (1 / latency) when there is overlap
    - in real processors there is always overlap
  - good metric for fixed amount of time (maximize work)

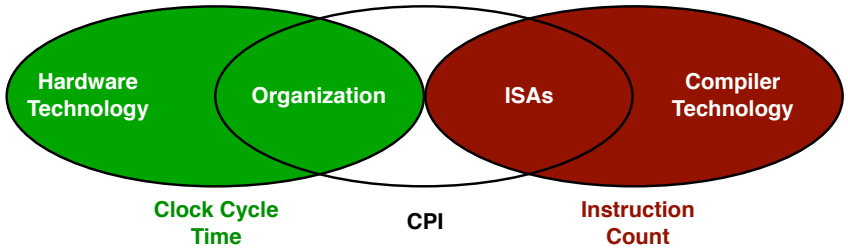


- comparing performance
  - A is N times faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = N
  - A is X% faster than B if and only if:
    - perf(A)/perf(B) = time(B)/time(A) = 1 + X/100

CPU time (the "best" metric)

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

- 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:



Encodings can be more complex (but fundamentally do the same thing)

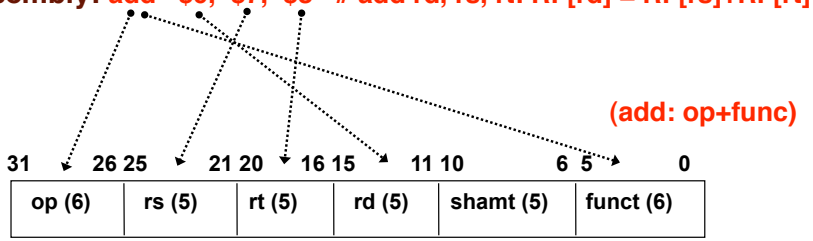
❑ 6-instruction processor:

Add instruction: **0010 $ra_3ra_2ra_1ra_0$ $rb_3rb_2rb_1rb_0$ $rc_3rc_2rc_1rc_0$**

Add Ra, Rb, Rc—specifies the operation *RF[a]=RF[b] + RF[c]*

❑ MIPS processor:

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

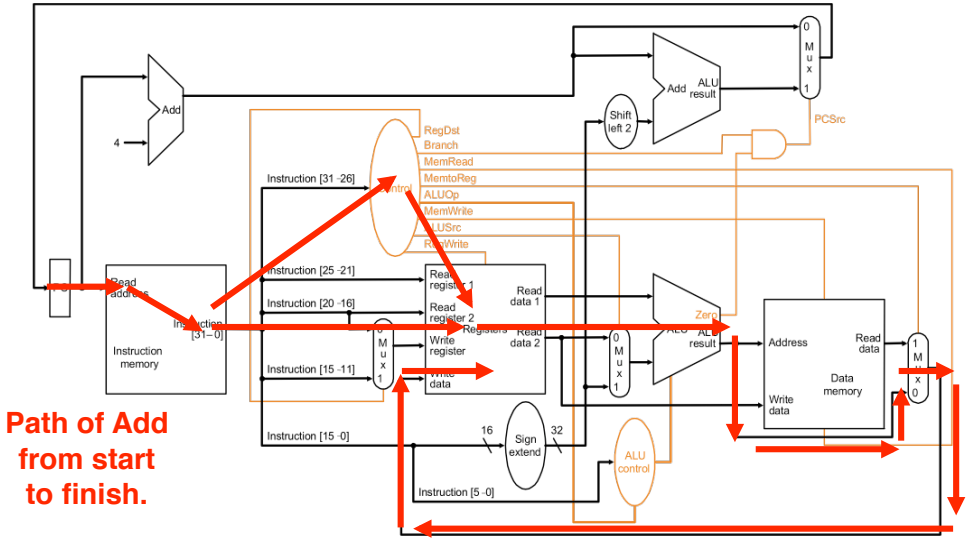


Machine:

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

Instruction Encoding

More complex instruction encodings, same path through datapath…



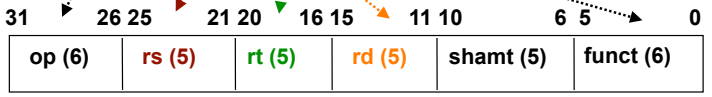
Path of Add from start to finish.

Review: MIPS R-Type

- R-type: All operands are in registers

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

(add: op+func)



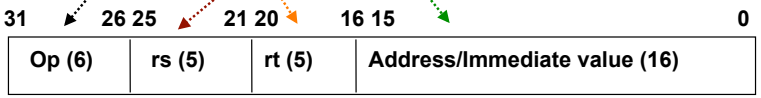
Machine:

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

Review: MIPS I-Type (arithmetic)

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

Example: addi $s2, $s1, 128 # addi rt, rs, Imm
RF[18] = RF[17]+128

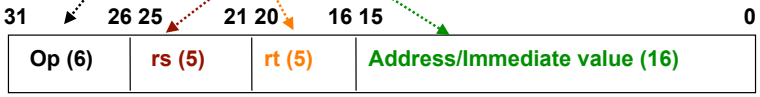


B:	001000	10001	10010	0000000010000000
D:	8	17	18	128

Review: MIPS I-Type (load/store)

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

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



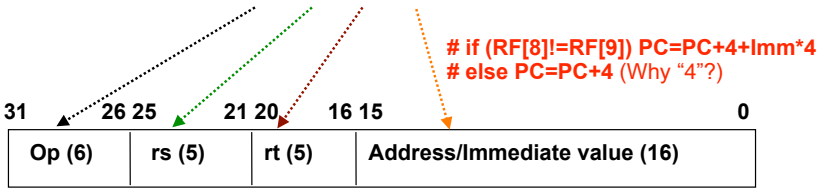
B:	100011	01000	10011	0000000000100000
D:	35	8	19	32

How about load the next word in memory?

Review: MIPS I-Type (branch)

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

Example: Again: bne $t0, $t1, Again

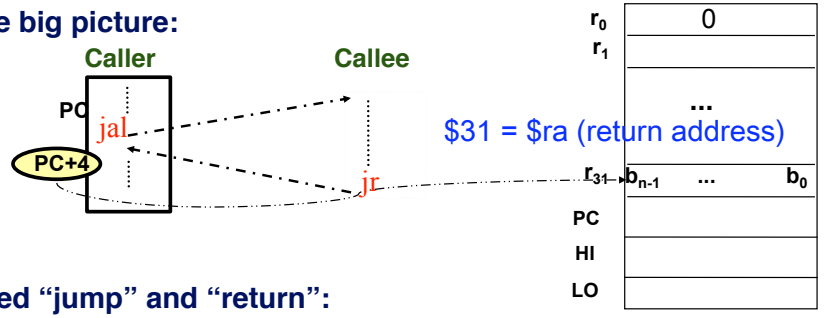


B:	00101	01000	01001	1111111111111111
D:	5	8	9	-1

PC-relative addressing

Procedure Handling (in MIPS)

- 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]

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

Procedure call essentials: Good Strategy

- Caller at call time
  - put arguments in $a0..$a4
  - save any caller-save temporaries
  - jal ..., $ra
- Callee at entry
  - allocate all stack space
  - save $ra, $fp + $s0..$s7 if necessary
- Callee at exit
  - restore $ra, $fp + $s0..$s7 if used
  - deallocate all stack space
  - put return value in $v0
- Caller after return
  - retrieve return value from $v0
  - restore any caller-save temporaries

do most work at callee entry/exit

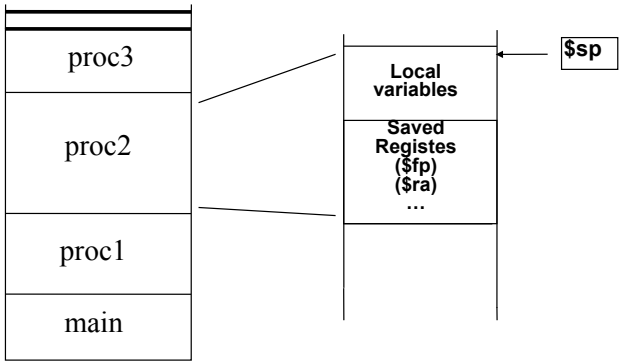
most of the work

Nested procedures

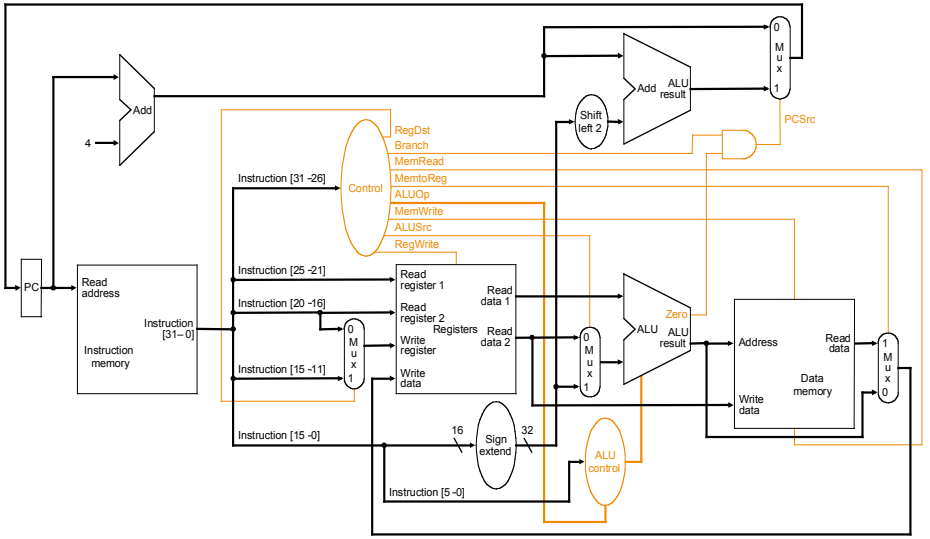
- Use the stack to save needed data

```
main {
…
  proc1
…}

proc1 {
…
  proc2
…}

proc2 {
…
  proc3
…}
```

Snap shots of stack



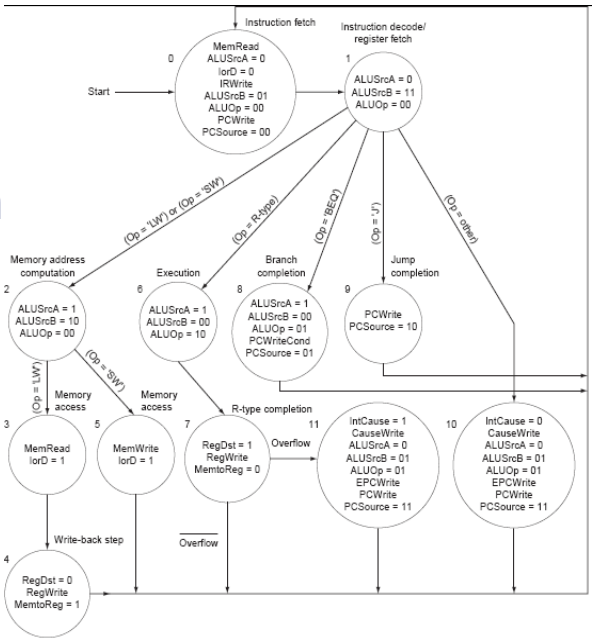
A Single Cycle Datapath



A (better performing) multi-cycle datapath

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Mem[PC], PC = PC + 4			
Instruction decode/register fetch	A =RF [IR[25:21]], B = RF [IR[20:16]], ALUOut = PC + (sign-extend (IR[1:-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15:0])	if (A =B) then PC = ALUOut	PC = PC [31:28] | (IR[25:0]<<2)
Memory access or R-type completion	RF [IR[15:11]] = ALUOut	Load: MDR = Mem[ALUOut] or Store: Mem[ALUOut]= B		
Memory read completion		Load: RF[IR[20:16]] = MDR		

Finite state diagram for MIPS multi-cycle datapath



MIPS multi-cycle datapath