EE488
Computer Architecture
Lecture 4: MIPS ISA

2024 Summer

Today's Lecture

- **Quick Review of Last Lecture**

- **Basic ISA Decisions and Design**

- **Announcements**

- **Operations**

- **Instruction Sequencing**

- **Delayed Branch**

- **Procedure Calling**

Quick Review of Last Lecture

Comparing Number of Instructions

Code sequence for (C = A + B) for four classes of instruction sets:

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R1,B	Load R2,B
Add	Store C	Store C, R1	Add R3,R1,R2
Pop C			Store C,R3

$$ExecutionTime = \frac{1}{Performance} = Instructions \times \frac{Cycles}{Instruction} \times \frac{Seconds}{Cycle}$$

General Purpose Registers Dominate

- **1975-2002 all machines use general purpose registers**

- **Advantages of registers**

  - **Registers are faster than memory**

  - **Registers compiler technology has evolved to efficiently generate code for register files**

    - **E.g., (A*B) – (C*D) – (E*F) can do multiplies in any order vs. stack**

  - **Registers can hold variables**

    - **Memory traffic is reduced, so program is sped up (since registers are faster than memory)**

  - **Code density improves (since register named with fewer bits than memory location)**

  - **Registers imply operand locality**

Operand Size Usage
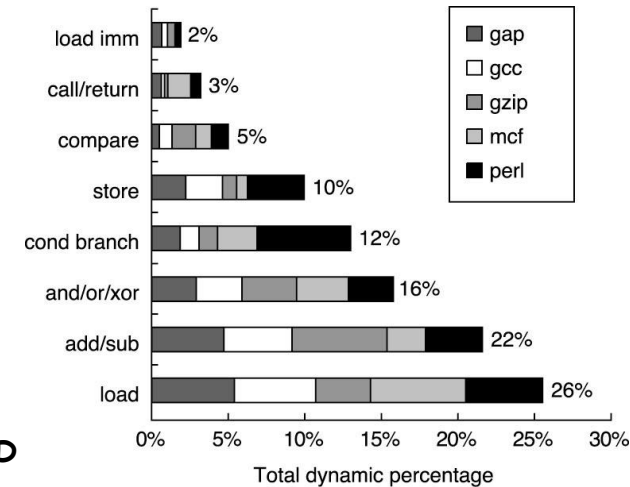

Frequency of reference by size
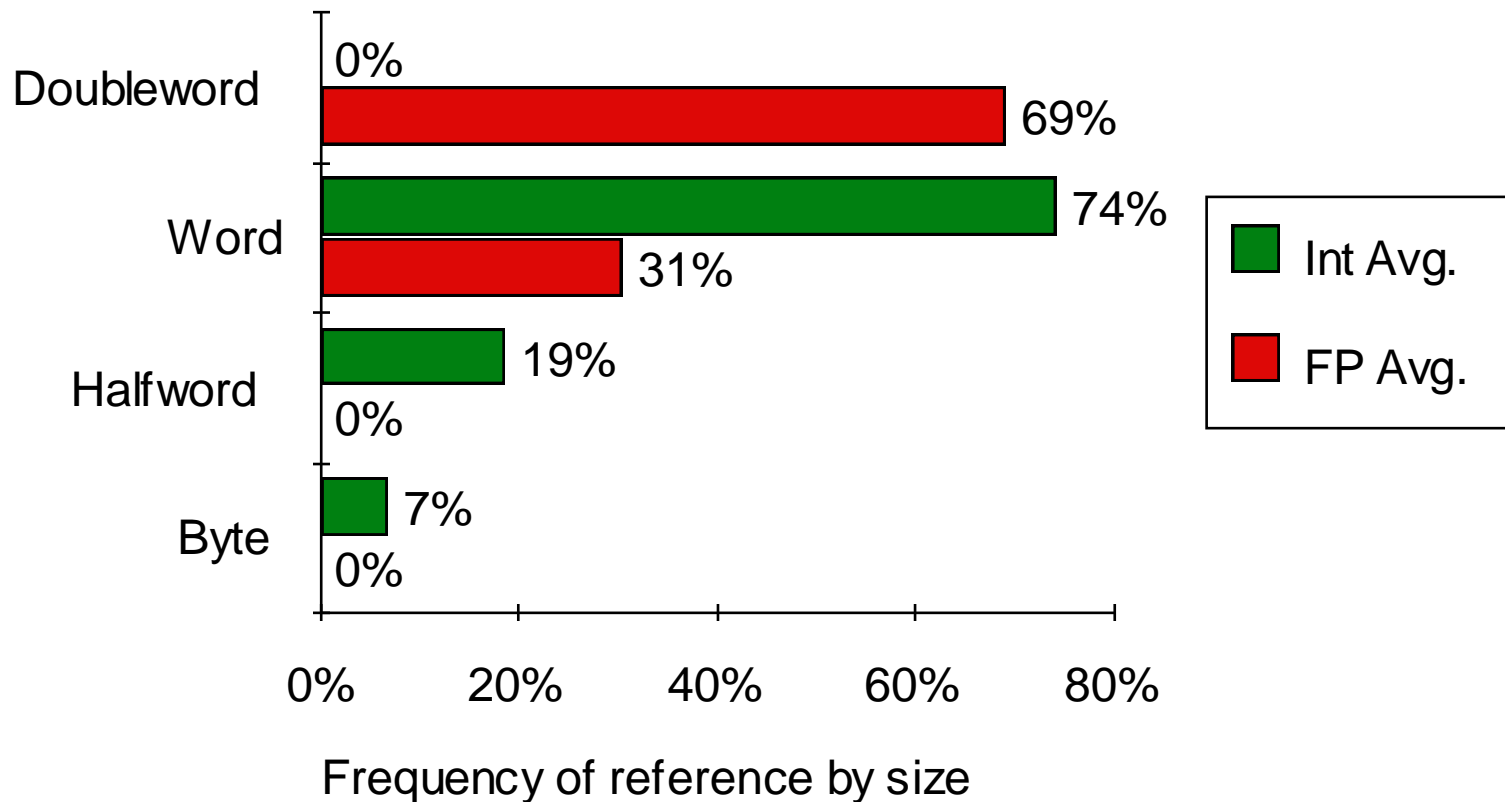
- Support for these data sizes and types:
    8-bit, 16-bit, 32-bit integers and
    32-bit and 64-bit IEEE 754 floating point numbers

Typical Operations (little change since 1960)

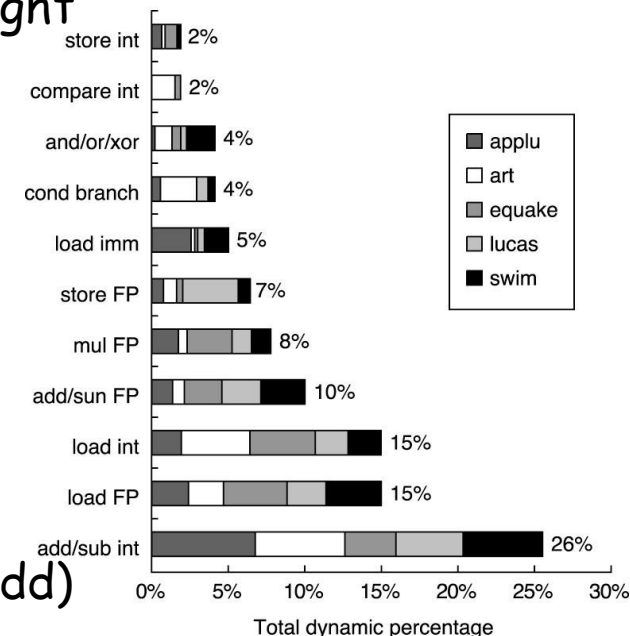|	
Data Movement	Load (from memory)
	Store (to memory)
	memory-to-memory move
	register-to-register move
	input (from I/O device)
	output (to I/O device)
	push, pop (to/from stack)
Arithmetic	integer (binary + decimal) or FP
	Add, Subtract, Multiply, Divide
Shift	shift left/right, rotate left/right
Logical	not, and, or, set, clear
Control (Jump/Branch)	unconditional, conditional
Subroutine Linkage	call, return
Interrupt	trap, return
Synchronization	test & set (atomic r-m-w)
String	search, translate
Graphics (MMX)	parallel subword ops (4 16bit add)

Chart 1 (legend: gap, gcc, gzip, mcf, perl):

Operation	Total dynamic percentage
load imm	2%
call/return	3%
compare	5%
store	10%
cond branch	12%
and/or/xor	16%
add/sub	22%
load	26%

Chart 2 (legend: applu, art, equake, lucas, swim):

| Operation	Total dynamic percentage
store int	2%
compare int	2%
and/or/xor	4%
cond branch	4%
load imm	5%
store FP	7%
mul FP	8%
add/sun FP	10%
load int	15%
load FP	15%
add/sub int	26%

Addressing Modes

- Addressing modes specify a constant, a register, or a location in memory

  - **Register**   `add r1, r2`   `r1 <- r1+r2`
  - **Immediate**   `add r1, #5`   `r1 <- r1+5`
  - **Direct**   `add r1, (0x200)`   `r1 <- r1+M[0x200]`
  - **Register indirect** `add r1, (r2)`   `r1 <- r1+M[r2]`
  - **Displacement**   `add r1, 100(r2)`   `r1 <- r1+M[r2+100]`
  - **Indexed**   `add r1, (r2+r3)`   `r1 <- r1+M[r2+r3]`
  - **Scaled**   `add r1, (r2+r3*4)`   `r1 <- r1+M[r2+r3*4]`
  - **Memory indirect**   `add r1, @(r2)`   `r1 <- r1+M[M[r2]]`
  - **Auto-increment**   `add r1, (r2)+`   `r1 <- r1+M[r2], r2++`
  - **Auto-decrement**   `add r1, -(r2)`   `r2--, r1 <- r1+M[r2]`

- Complicated modes reduce instruction count at the cost of complex implementations

Instruction Sequencing

- **The next instruction to be executed is typically implied**
    - **Instructions execute sequentially**
    - **Instruction sequencing increments a Program Counter**

Instruction 1
Instruction 2
Instruction 3
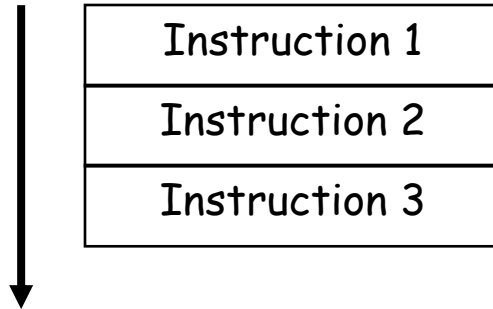
- **Sequencing flow is disrupted conditionally and unconditionally**
    - **The ability of computers to test results and conditionally instructions is one of the reasons computers have become so useful**

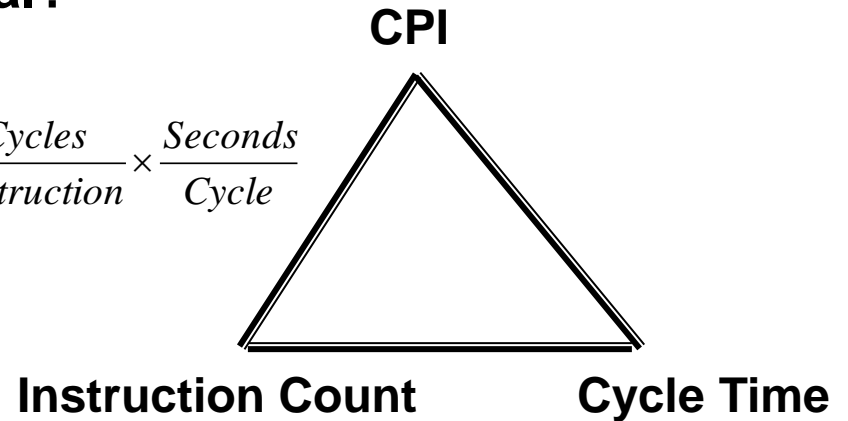| Instruction 1 |
| :---: |
| Instruction 2 |
| Conditional Branch |
| Instruction 4 |
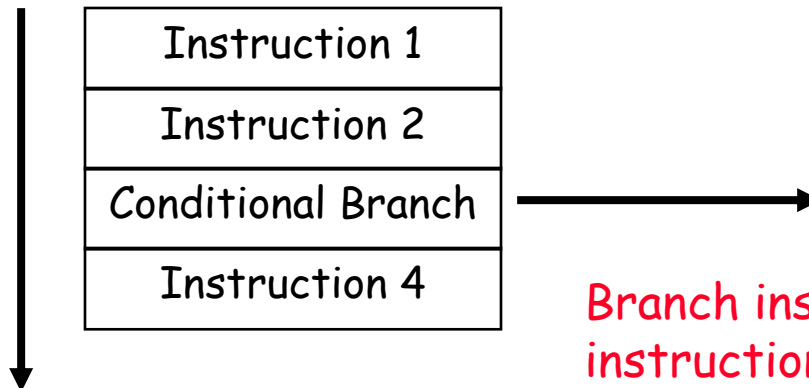
Branch instructions are ~20% of all instructions executed

Instruction Set Design Metrics

- **Static Metrics**
  - **How many bytes does the program occupy in memory?**

- **Dynamic Metrics**
  - **How many instructions are executed?**
  - **How many bytes does the processor fetch to execute the program?**
  - **How many clocks are required per instruction?**
  - **How "lean" a clock is practical?**

- $ExecutionTime = \dfrac{1}{Performance} = Instructions \times \dfrac{Cycles}{Instruction} \times \dfrac{Seconds}{Cycle}$

CPI

Instruction Count   **Cycle Time**

MIPS R2000 / R3000 Registers

- **Programmable storage**

r0	0
r1	
°	
°	
°	
r31	

PC	

lo	
hi	
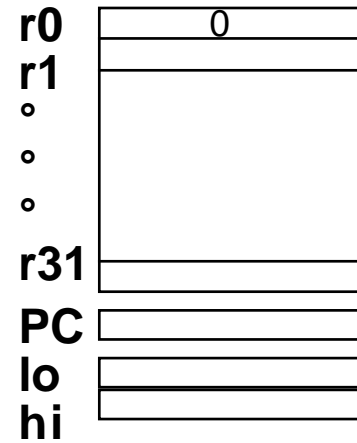
MIPS Addressing Modes/Instruction Formats

• **All instructions 32 bits wide**

Register (direct)

op	rs	rt	rd	

register

* rs - source reg; rt - second reg; rd - destination reg

Immediate

op	rs	rt	immed

Base+index

op	rs	rt	immed

register + → **Memory**

PC-relative

op	rs	rt	immed
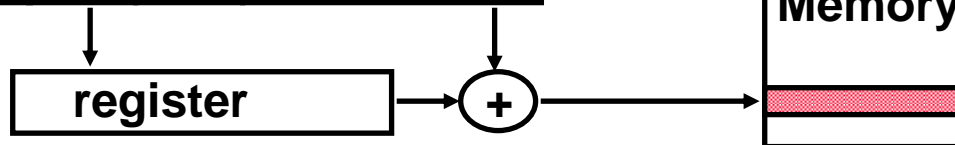
PC + → **Memory**
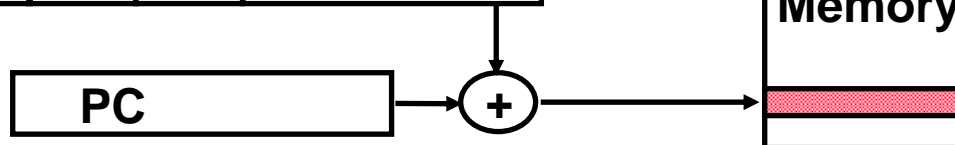
MIPS R2000 / R3000 Operation Overview

- **Arithmetic logical**

- **Add, AddU, Sub,   SubU, And, Or, Xor, Nor, SLT, SLTU**

- **AddI, AddIU, SLTI, SLTIU, AndI, OrI, XorI, LUI**

- **SLL, SRL, SRA, SLLV, SRLV, SRAV**

- **Memory Access**

- **LB, LBU, LH, LHU, LW, LWL,LWR**
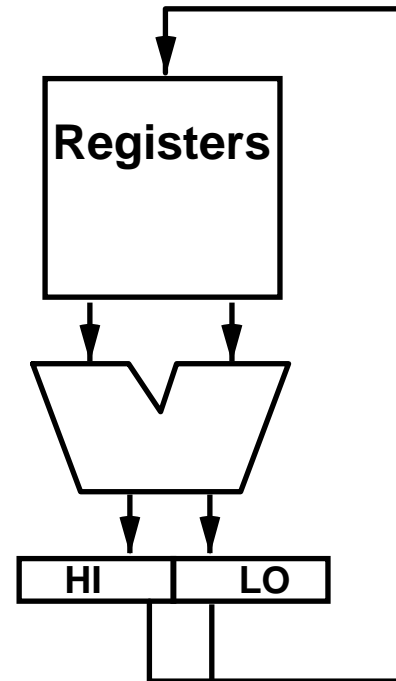
- **SB, SH, SW, SWL, SWR**

Multiply / Divide

- **Start multiply, divide**
  - **MULT rs, rt**
  - **MULTU rs, rt**
  - **DIV rs, rt**
  - **DIVU rs, rt**
- **Move result from multiply, divide**
  - **MFHI rd**
  - **MFLO rd**
- **Move to HI or LO**
  - **MTHI rd**
  - **MTLO rd**

Registers

HI   LO
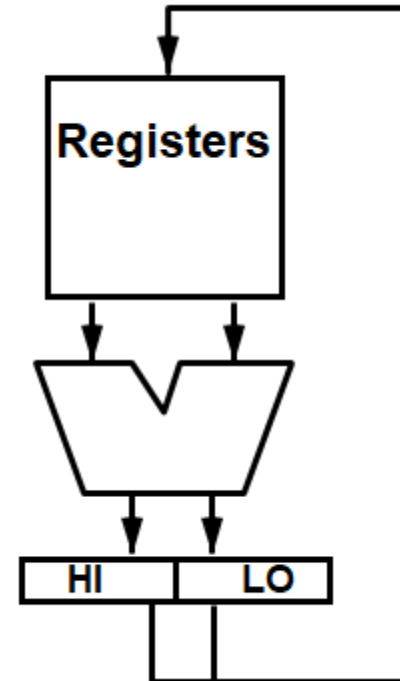
Multiply / Divide

- **Start multiply, divide**
    - **MULT rs, rt**
- **Move to HI or LO**
    - **MTHI rd**
    - **MTLO rd**
- **Why not Third field for destination? (Hint: how many clock cycles for multiply or divide vs. add?)**

MIPS arithmetic instructions

Instruction	Example	Meaning	Comments
add	add $1,$2,$3	$1 = $2 + $3	3 operands; exception possible
subtract	sub $1,$2,$3	$1 = $2 − $3	3 operands; exception possible
add immediate	addi $1,$2,100	$1 = $2 + 100	+ constant; exception possible
add unsigned	addu $1,$2,$3	$1 = $2 + $3	3 operands; no exceptions
subtract unsigned	subu $1,$2,$3	$1 = $2 − $3	3 operands; no exceptions
add imm. unsign.	addiu $1,$2,100	$1 = $2 + 100	+ constant; no exceptions
multiply	mult $2,$3	Hi, Lo = $2 x $3	64-bit signed product
multiply unsigned	multu $2,$3	Hi, Lo = $2 x $3	64-bit unsigned product
divide	div $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Lo = quotient, Hi = remainder
divide unsigned	divu $2,$3	Lo = $2 ÷ $3, Hi = $2 mod $3	Unsigned quotient & remainder
Move from Hi	mfhi $1	$1 = Hi	Used to get copy of Hi
Move from Lo	mflo $1	$1 = Lo	Used to get copy of Lo

MIPS logical instructions

Instruction	Example	Meaning	Comment
and	and $1,$2,$3	$1 = $2 & $3	3 reg. operands; Logical AND
or	or $1,$2,$3	$1 = $2 | $3	3 reg. operands; Logical OR
xor	xor $1,$2,$3	$1 = $2 ⊕ $3	3 reg. operands; Logical XOR
nor	nor $1,$2,$3	$1 = ~($2 |$3)	3 reg. operands; Logical NOR
and immediate	andi $1,$2,10	$1 = $2 & 10	Logical AND reg, constant
or immediate	ori $1,$2,10	$1 = $2 | 10	Logical OR reg, constant
xor immediate	xori $1, $2,10	$1 = ~$2 &~10	Logical XOR reg, constant
shift left logical	sll $1,$2,10	$1 = $2 << 10	Shift left by constant
shift right logical	srl $1,$2,10	$1 = $2 >> 10	Shift right by constant
shift right arithm.	sra $1,$2,10	$1 = $2 >> 10	Shift right (sign extend)
shift left logical	sllv $1,$2,$3	$1 = $2 << $3	Shift left by variable
shift right logical	srlv $1,$2, $3	$1 = $2 >> $3	Shift right by variable
shift right arithm.	srav $1,$2, $3	$1 = $2 >> $3	Shift right arith. by variable

MIPS data transfer instructions

Instruction	Comment
SW 500(R4), R3	Store word
SH 502(R2), R3	Store half
SB 41(R3), R2	Store byte
LW R1, 30(R2)	Load word
LH R1, 40(R3)	Load halfword
LHU R1, 40(R3)	Load halfword unsigned
LB R1, 40(R3)	Load byte
LBU R1, 40(R3)	Load byte unsigned

LUI R1, 40   **Load Upper Immediate (16 bits shifted left by 16)**

* 40(dec) -> 28 Hex
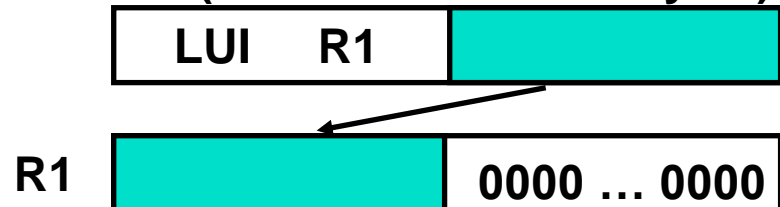
Methods of Testing Condition

- **Condition Codes**

  Processor status bits are set as a side-effect of arithmetic instructions (possibly on Moves) or explicitly by compare or test instructions.

  ex: add r1, r2, r3

  bz label

- **Condition Register**

  Ex: cmp r1, r2, r3

  bgt r1, label

- **Compare and Branch**

  Ex: bgt r1, r2, label

Condition Codes

Setting CC as side effect can reduce the # of instructions

```
X:        .                          X:        .
          .              vs.                   .
          .                                    .
   SUB  r0, #1, r0                      SUB  r0, #1, r0
   BRP  X                               CMP  r0, #0
                                        BRP  X
```

But also has disadvantages:

--- not all instructions set the condition codes;
which do and which do not often confusing!
e.g., shift instruction sets the carry bit

--- dependency between the instruction that sets the CC and the one
that tests it: to overlap their execution, may need to separate them
with an instruction that does not change the CC

ifetch	read	compute	write

Old CC read   New CC computed

ifetch	read	compute	write
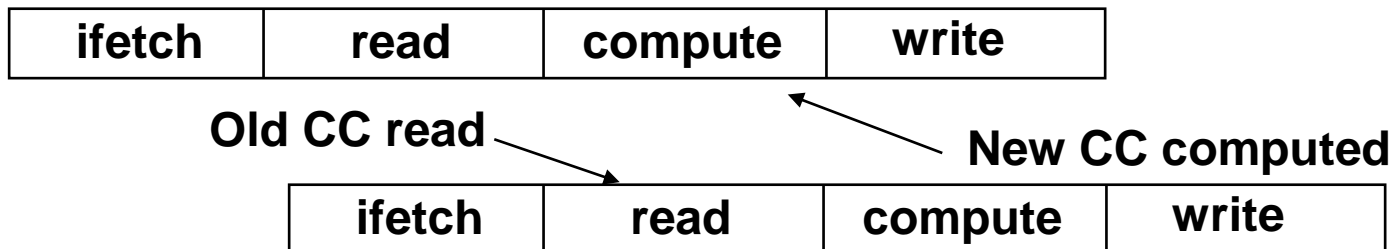
Compare and Branch

- **Compare and Branch**

  - **BEQ rs, rt, offset   if R[rs] == R[rt] then PC-relative branch**

  - **BNE rs, rt, offset   <>0**

- **Compare to zero and Branch**

  - **BLEZ rs, offset   if R[rs] <= 0 then PC-relative branch**

  - **BGTZ rs, offset   >0**

  - **BLT   <0**

  - **BGEZ   >=0**

  - **BLTZAL rs, offset   if R[rs] < 0 then branch and link (into R 31)**

  - **BGEZAL   >=0**

- **Remaining set of compare and branch take two instructions**

- **Almost all comparisons are against zero!**

MIPS jump, branch, compare instructions

Instruction	*Example*	*Meaning*
branch on equal	beq $1,$2,100	if ($1 == $2) go to PC+4+100
		Equal test; PC relative branch
branch on not eq.	bne $1,$2,100	if ($1!= $2) go to PC+4+100
		Not equal test; PC relative
set on less than	slt $1,$2,$3	if ($2 < $3) $1=1; else $1=0
		Compare less than; 2's comp.
set less than imm.	slti $1,$2,100	if ($2 < 100) $1=1; else $1=0
		Compare < constant; 2's comp.
set less than uns.	sltu $1,$2,$3	if ($2 < $3) $1=1; else $1=0
		Compare less than; natural numbers
set l. t. imm. uns.	sltiu $1,$2,100	if ($2 < 100) $1=1; else $1=0
		Compare < constant; natural numbers
jump	j 10000	go to 10000
		Jump to target address
jump register	jr $31	go to $31
		For switch, procedure return
jump and link	jal 10000	$31 = PC + 4; go to 10000
		For procedure call

Signed vs. Unsigned Comparison

R1= 0…00 0000 0000 0000 0001$_2$

R2= 0…00 0000 0000 0000 0010$_2$

R3= 1…11 1111 1111 1111 1111$_2$

° **After executing these instructions:**

```
slt  r4,r2,r1 ;
```
if (r2 < r1) r4=1; else r4=0

```
slt  r5,r3,r1 ;
```
if (r3 < r1) r5=1; else r5=0

```
sltu r6,r2,r1 ;
```
if (r2 < r1) r6=1; else r6=0

```
sltu r7,r3,r1 ;
```
if (r3 < r1) r7=1; else r7=0

° **What are values of registers r4 - r7? Why?**

r4 =   ; r5 =   ; r6 =   ; r7 =   ;

Calls: Why Are Stacks So Great?

Stacking of Subroutine Calls & Returns and Environments:

A:
    CALL B

    B:
       CALL C

       C:
         RET

     RET

A		
A	B	
A	B	C
A	B	
A		
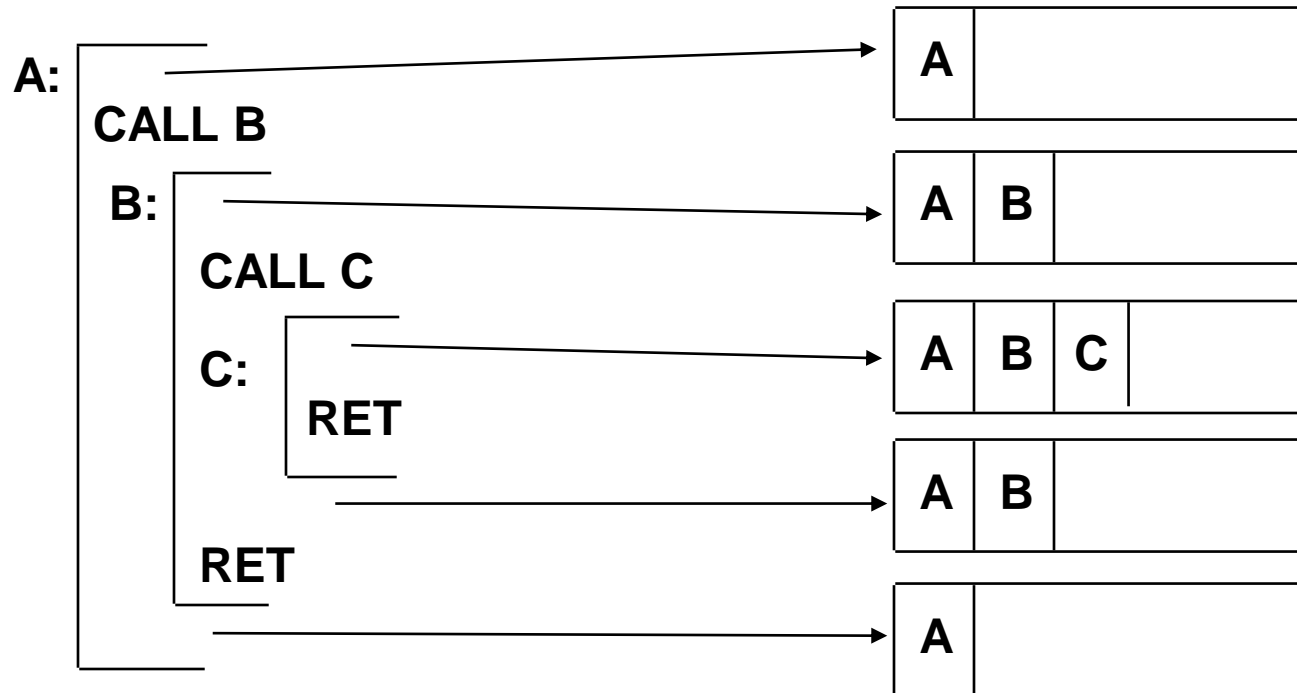
Some machines provide a memory stack as part of the architecture (e.g., VAX)

Sometimes stacks are implemented via software convention (e.g., MIPS)

Memory Stacks

Useful for stacked environments/subroutine call & return even if operand stack not part of architecture

Stacks that Grow Up vs. Stacks that Grow Down:
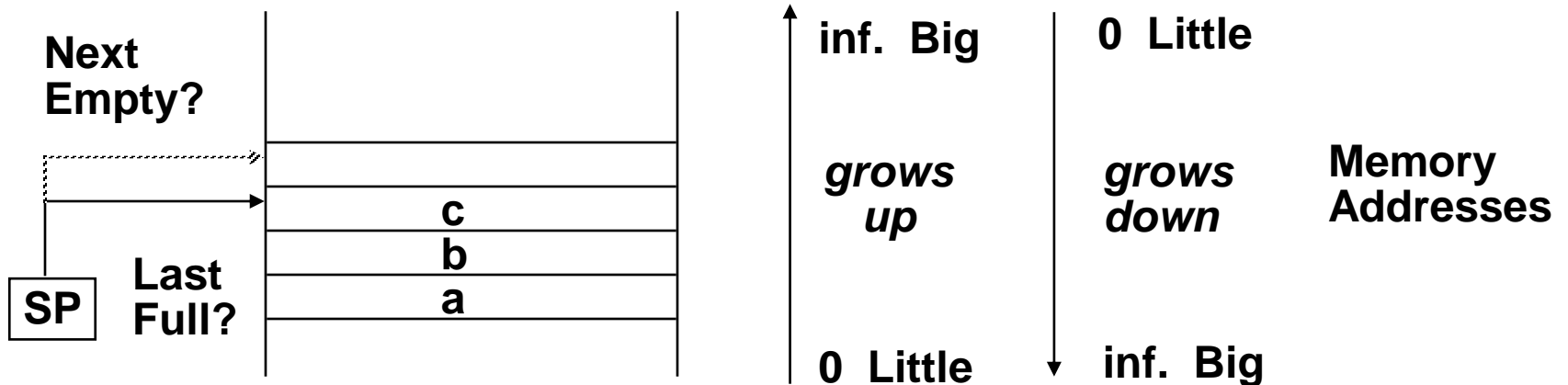
How is empty stack represented?

Little --> Big/Last Full

POP: Read from Mem(SP)
            Decrement SP

PUSH: Increment SP
            Write to Mem(SP)

Little --> Big/Next Empty

POP: Decrement SP
            Read from Mem(SP)

PUSH: Write to Mem(SP)
            Increment SP

Call-Return Linkage: Stack Frames

High Mem

ARGS
Callee Save Registers   **(old FP, RA)**
Local Variables

Reference args and local variables at fixed (positive) offset from FP

FP →

SP →

Grows and shrinks during expression evaluation

Low Mem
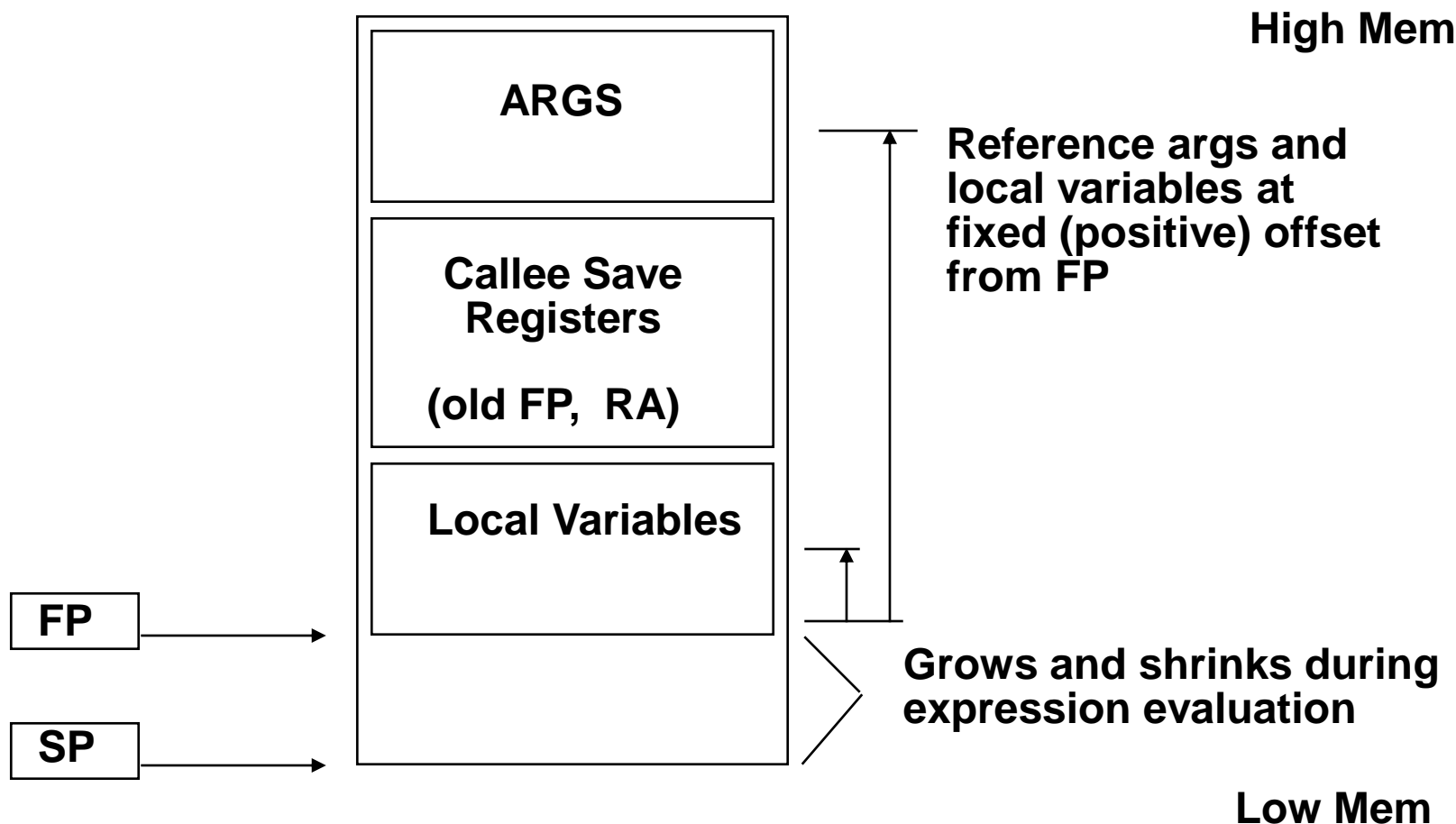
° **Many variations on stacks possible (up/down, last pushed / next)**

° **Block structured languages contain link to lexically enclosing frame**

° **Compilers normally keep scalar variables in registers, not memory!**

Call-Return Linkage: Stack Frames
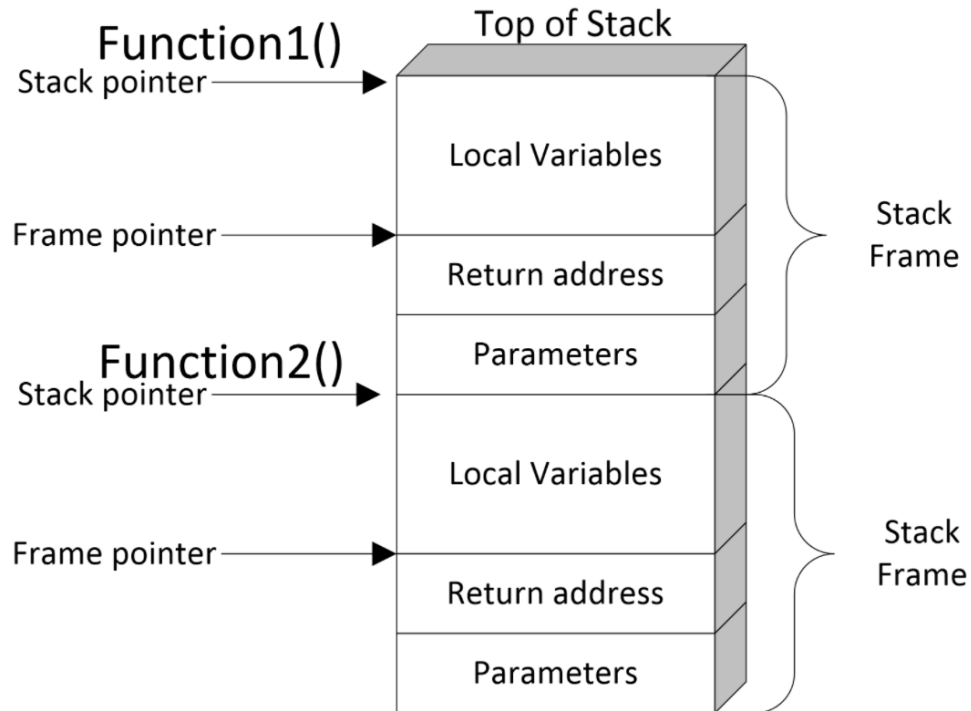
MIPS: Software conventions for Registers

0	zero	constant 0
1	at	reserved for assembler

2	v0	expression evaluation &
3	v1	function results

4	a0	arguments
5	a1	
6	a2	
7	a3	

8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	

16	s0	callee saves
. . .		(caller can clobber)
23	s7	

24	t8	temporary (cont'd)
25	t9	

26	k0	reserved for OS kernel
27	k1	

28	gp	Pointer to global area
29	sp	Stack pointer
30	fp	frame pointer

31	ra	Return Address (HW)

Plus a 3-deep stack of mode bits.

Example in C: swap

```
swap(int v[], int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

° **Assume swap is called as a procedure**

° **Assume temp is register $15; arguments in $a1, $a2; $16 is scratch reg:**

° **Write MIPS code**

swap: MIPS

swap:

```
addiu  $sp,$sp, –4      ; create space on stack
sw     $16, 4($sp)      ; callee saved register put onto stack
sll    $t2, $a2,2       ; mulitply k by 4
addu   $t2, $a1,$t2     ; address of v[k]
lw     $15, 0($t2)      ; load v[k]
lw     $16, 4($t2)      ; load v[k+1]
sw     $16, 0($t2)      ; store v[k+1] into v[k]
sw     $15, 4($t2)      ; store old value of v[k] into v[k+1]
lw     $16, 4($sp)      ; callee saved register restored from stack
addiu  $sp,$sp, 4       ; restore top of stack
jr     $31              ; return to place that called swap
```

Delayed Branches

```
        li    r3, #7

        sub   r4, r4, 1

        bz    r4, LL

        addi  r5, r3, 1

        subi  r6, r6, 2

   LL:  slt   r1, r3, r5
```

° **In the "Raw" MIPS the instruction after the branch is executed even when the branch is taken?**

  - **This is hidden by the assembler for the MIPS "virtual machine"**
  - **allows the compiler to better utilize the instruction pipeline (???)**

Branch & Pipelines

Time →

```
li r3, #7          [ execute ]

sub r4, r4, 1      [ ifetch | execute ]

bz r4, LL               [ ifetch | execute ]   Branch

addi r5, r3, 1              [ ifetch | execute ]   Delay Slot

LL: slt  r1, r3, r5   Branch Target      [ ifetch | execute ]
```
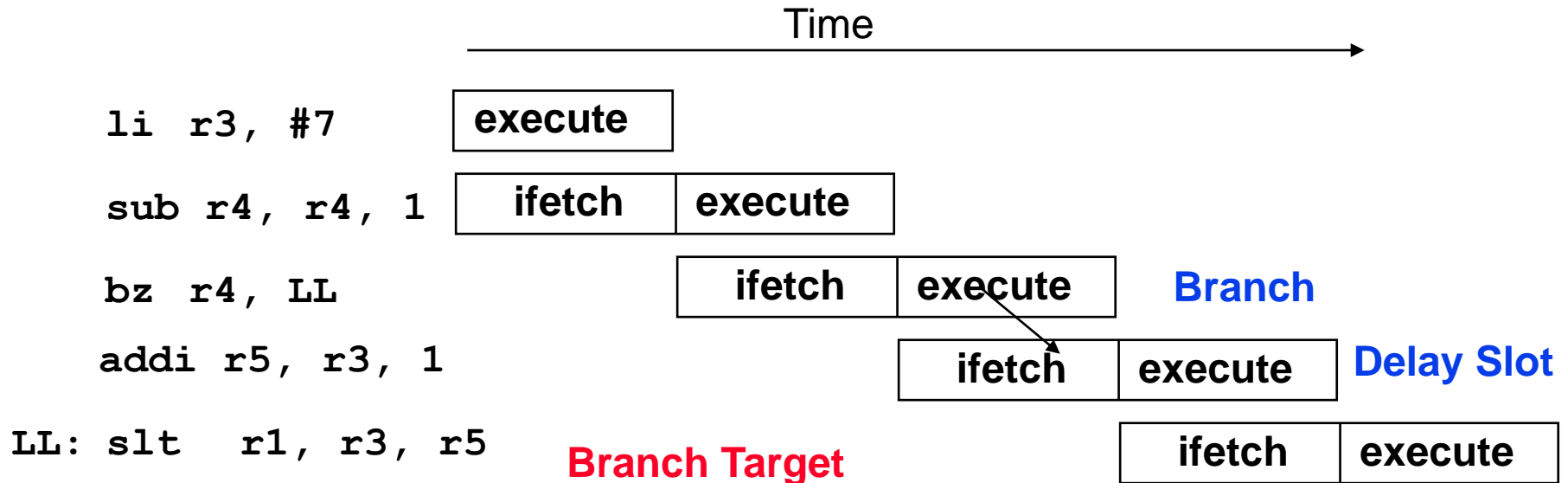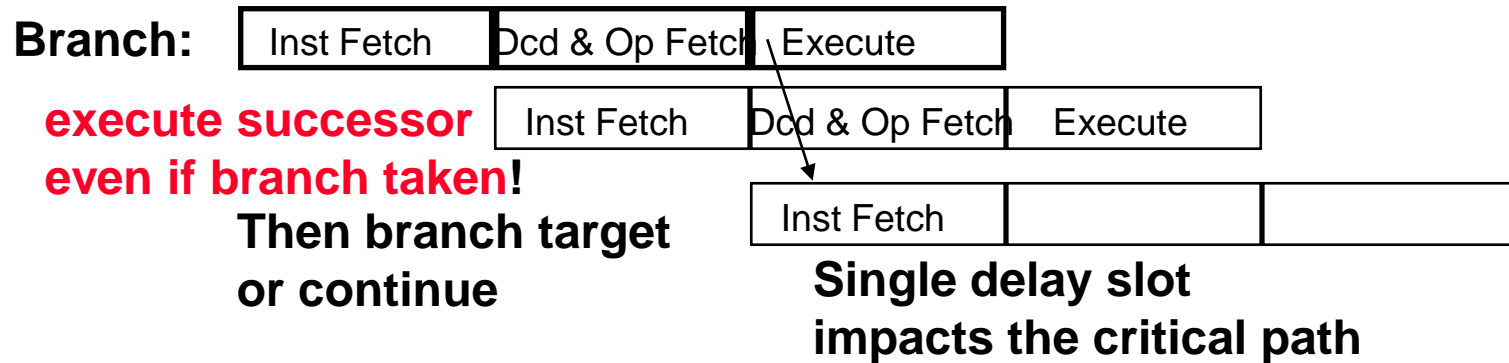
By the end of Branch instruction, the CPU knows whether or not the branch will take place.

However, it will have fetched the next instruction by then, regardless of whether or not a branch will be taken.

Why not execute it?

Filling Delayed Branches

Branch:

Inst Fetch	Dcd & Op Fetch	Execute

**execute successor
even if branch taken!**

Inst Fetch	Dcd & Op Fetch	Execute

**Then branch target
or continue**

Inst Fetch		

**Single delay slot
impacts the critical path**

•**Compiler can fill a single delay slot
with a useful instruction 50% of the
time.**

• **try to move down from above
jump**

•**move up from target, if safe**

```
add r3, r1, r2

sub r4, r4, 1

bz  r4, LL

NOP

...

LL:    add rd, ...
```

Is this violating the ISA abstraction?

Standard and Delayed Interpretation

PC

```
add  rd, rs, rt      R[rd] <- R[rs] + R[rt];
                     PC <- PC + 4;

beq  rs, rt, offset  if  R[rs] == R[rt] then PC <- PC + SX(offset)
                                    else PC <- PC + 4;

sub  rd, rs, rt      . . .

            . . .
L1:    target
```

PC
nPC

```
add  rd, rs, rt      R[rd] <- R[rs] + R[rt];
                     PC <- nPC;   nPC <- nPC + 4;

beq  rs, rt, offset  if  R[rd] == R[rt] then nPC <- nPC + SX(offset)
                                    else nPC <- nPC + 4;

                     PC <- nPC

sub  rd, rs, rt      . . .

            . . .
L1:    target                          Delayed Loads?
```
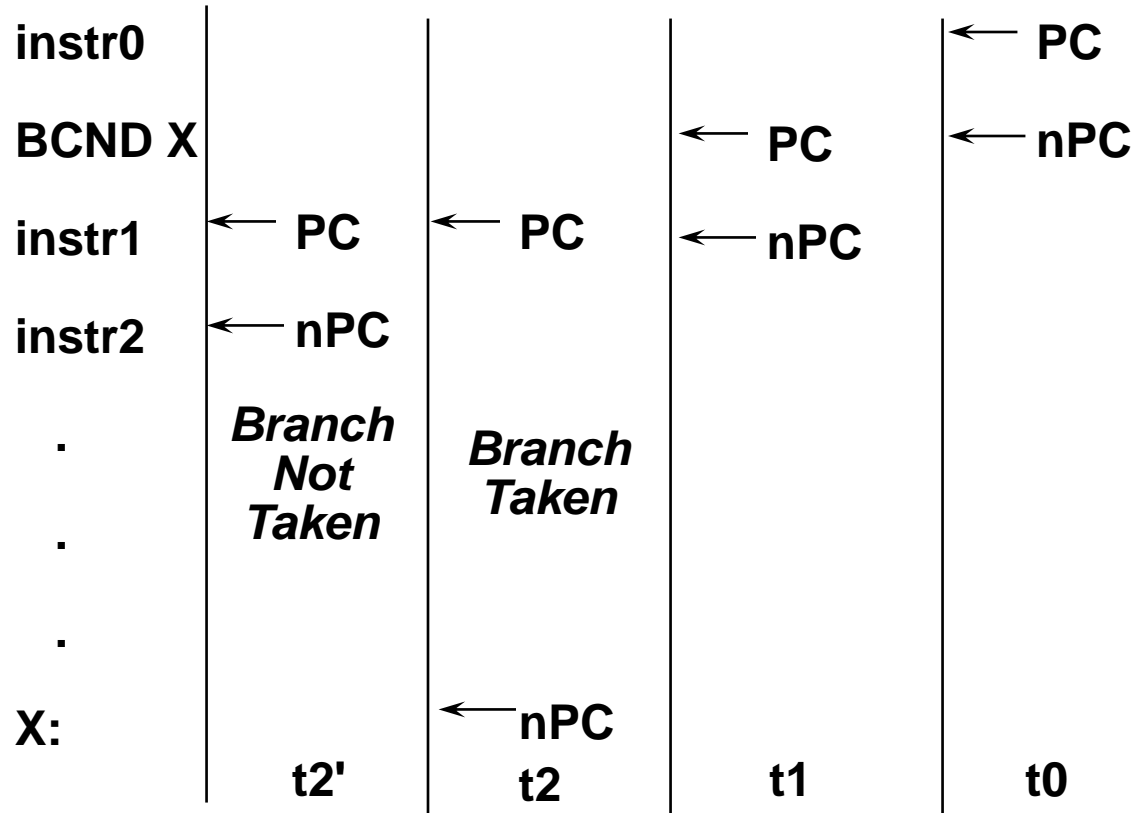
Delayed Branches (cont.)

Execution History


```
instr0                                          ← PC
BCND X                        ← PC             ← nPC
instr1    ← PC    ← PC        ← nPC
instr2    ← nPC
  .       Branch
  .       Not      Branch
          Taken    Taken
  .
  .
X:                 ← nPC
          t2'      t2          t1               t0
```

Branches are the bane (or pain!) of pipelined machines
Delayed branches complicate the compiler slightly, but make pipelining
   easier to implement and more effective
Good strategy to move some complexity to compile time

Details of the MIPS instruction set

- **Register zero always has the value zero (even if you try to write it)**

- **Branch and jump instructions put the return address PC+4 into the link register**

- **All instructions change all 32 bits of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)**

- **Immediate arithmetic and logical instructions are extended as follows:**

    - **logical immediates are zero extended to 32 bits**

    - **arithmetic immediates are sign extended to 32 bits**

- **The data loaded by the instructions lb and lh are extended as follows:**

    - **lbu, lhu are zero extended**

    - **lb, lh are sign extended**

- **Overflow can occur in these arithmetic and logical instructions:**

    - **add, sub, addi**

    - **it <u>cannot</u> occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu**

Other ISAs

- **Intel 8086/88 => 80286 => 80386 => 80486 => Pentium => P6**
  - **8086 few transistors to implement 16-bit microprocessor**
  - **tried to be somewhat compatible with 8-bit microprocessor 8080**
  - **successors added features which were missing from 8086 over next 15 years**
  - **product several different intel engineers over 10 to 15 years**
  - **Announced 1978**
- **VAX simple compilers & small code size =>**
  - **efficient instruction encoding**
  - **powerful addressing modes**
  - **powerful instructions**
  - **few registers**
  - **product of a single talented architect**
  - **Announced 1977**

Machine Examples: Address & Registers

Intel 8086	2^{20} x 8 bit bytes	acc, index, count, quot
	AX, BX, CX, DX	stack, string
	SP, BP, SI, DI	code,stack,data segment
	CS, SS, DS	
	IP, Flags	

		r15-- program counter
VAX 11	2^{32} x 8 bit bytes	r14-- stack pointer
	16 x 32 bit GPRs	r13-- frame pointer
		r12-- argument ptr

MC 68000	2^{24} x 8 bit bytes	
	8 x 32 bit GPRs	
	7 x 32 bit addr reg	
	1 x 32 bit SP	
	1 x 32 bit PC	

MIPS	2^{32} x 8 bit bytes	
	32 x 32 bit GPRs	
	32 x 32 bit FPRs	
	HI, LO, PC	

Details of the MIPS instruction set

- **Register zero <u>always</u> has the value <u>zero</u> (even if you try to write it)**

- **Branch/jump <u>and link</u> put the return addr. PC+4 into the link register (R31)**

- **All instructions change <u>all 32 bits</u> of the destination register (including lui, lb, lh) and all read all 32 bits of sources (add, sub, and, or, …)**

- **Immediate arithmetic and logical instructions are extended as follows:**
    - **logical immediates ops are zero extended to 32 bits**
    - **arithmetic immediates ops are sign extended to 32 bits (including addu)**

- **The data loaded by the instructions lb and lh are extended as follows:**
    - **lbu, lhu are zero extended**
    - **lb, lh are sign extended**

- **Overflow can occur in these arithmetic and logical instructions:**
    - **add, sub, addi**
    - **it <u>cannot</u> occur in addu, subu, addiu, and, or, xor, nor, shifts, mult, multu, div, divu**

Summary

- **Use general purpose registers with a load-store architecture: YES**

- **Provide at least 16 general purpose registers plus separate floating-point registers: 31 GPR & 32 FPR**

- **Support these addressing modes: displacement (with an address offset size of 12 to 16 bits), immediate (size 8 to 16 bits), and register deferred; : YES: 16 bits for immediate, displacement (disp=0 => register deferred)**

- **All addressing modes apply to all data transfer instructions : YES**

- **Use fixed instruction encoding if interested in performance and use variable instruction encoding if interested in code size : Fixed**

- **Support these data sizes and types: 8-bit, 16-bit, 32-bit integers and 32-bit and 64-bit IEEE 754 floating point numbers: YES**

- **Support these simple instructions, since they will dominate the number of instructions executed: load, store, add, subtract, move register-register, and, shift, compare equal, compare not equal, branch (with a PC-relative address at least 8-bits long), jump, call, and return: YES, 16b**

- **Aim for a minimalist instruction set: YES**

Summary: Salient features of MIPS R3000

- **32-bit fixed format inst** (3 formats)
- **32 32-bit GPR** (R0 contains zero) and 32 FP registers (and HI LO)
  - partitioned by software convention
- **3-address, reg-reg arithmetic instr.**
- **Single address mode for load/store:** base+displacement
  - no indirection
  - **16-bit immediate plus LUI**
- **Simple branch conditions**
  - compare against zero or two registers for =
  - no condition codes
- **Delayed branch**
  - execute instruction after the branch (or jump) even if the banch is taken (Compiler can fill a delayed branch with useful work about 50% of the time)