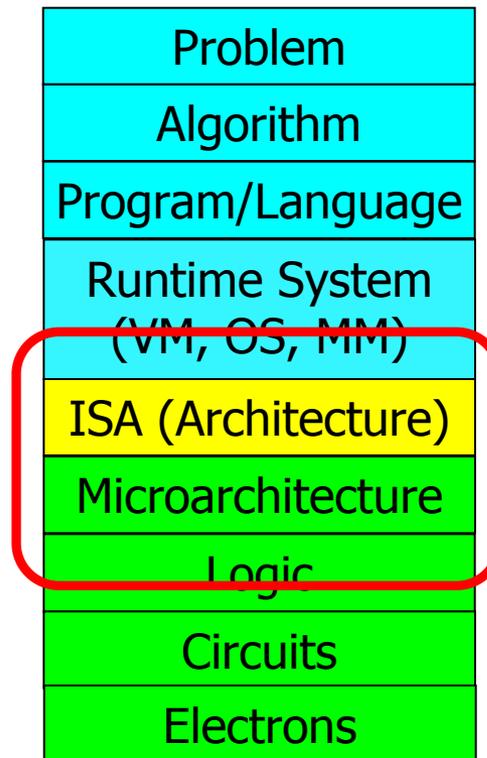


# 18-447

# MIPS ISA

James C. Hoe  
Dept of ECE, CMU  
January 19, 2012

# Levels of Transformation



# Terminologies

- ◆ Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- ◆ Instruction Set
  - the set of commands understood by the computer
- ◆ Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- ◆ Assembly Code
  - a collection of instructions expressed in “textual” format e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code (mostly true: compound instructions, address labels ....)

# What are specified/decided in an ISA?

- ◆ Data format and size
  - character, binary, decimal, floating point, negatives
- ◆ “Programmer Visible State”
  - memory, registers, program counters, etc.
- ◆ Instructions: how to transform the programmer visible state?
  - what to perform and what to perform next
  - where are the operands
- ◆ Instruction-to-binary encoding
- ◆ Protection and privileged operations

Very often you compromise immediate optimality for future scalability and compatibility

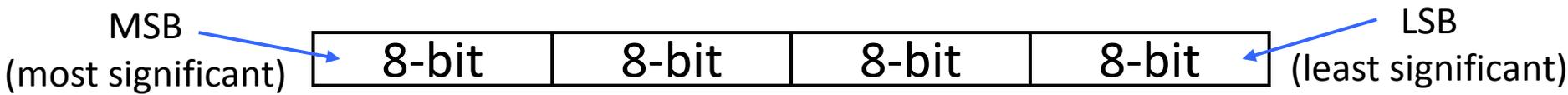


# Data Format

- ◆ Most things are 32 bits
  - instruction and data addresses
  - signed and unsigned integers
- ◆ Also 16-bit word and 8-bit word (aka byte)
- ◆ Floating-point numbers
  - IEEE standard 754
  - float: 8-bit exponent, 23-bit significand
  - double: 11-bit exponent, 52-bit significand

# Big Endian vs. Little Endian

- ◆ 32-bit signed or unsigned integer comprises 4 bytes



- ◆ On a byte-addressable machine . . . . .

## Little Endian

MSB			LSB
byte 3	byte 2	byte 1	byte 0
byte 7	byte 6	byte 5	byte 4
byte 11	byte 10	byte 9	byte 8
byte 15	byte 14	byte 13	byte 12
byte 19	byte 18	byte 17	byte 16

pointer points to the little end

## Big Endian

MSB			LSB
byte 0	byte 1	byte 2	byte 3
byte 4	byte 5	byte 6	byte 7
byte 8	byte 9	byte 10	byte 11
byte 12	byte 13	byte 14	byte 15
byte 16	byte 17	byte 18	byte 19

pointer points to the big end

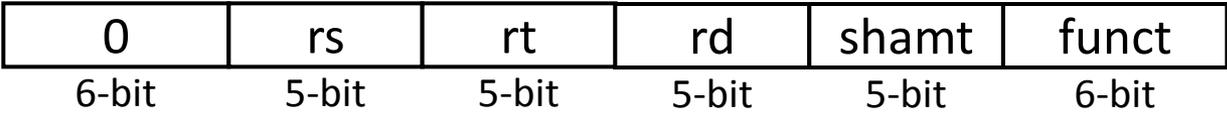
- ◆ What difference does it make?

check out `htonl()`, `ntohl()` in `in.h`

# Instruction Formats

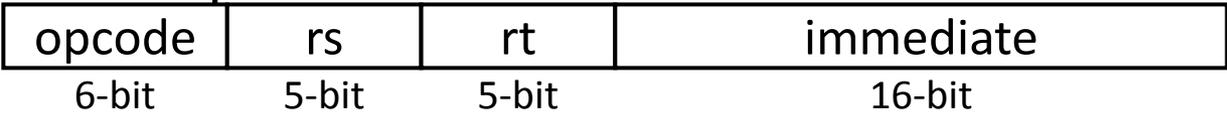
◆ 3 simple formats

- R-type, 3 register operands



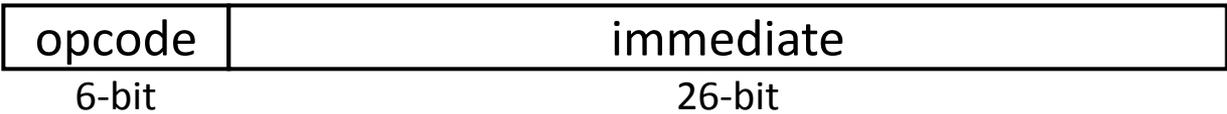
R-type

- I-type, 2 register operands and 16-bit immediate



I-type

- J-type, 26-bit immediate operand



J-type

◆ Simple Decoding

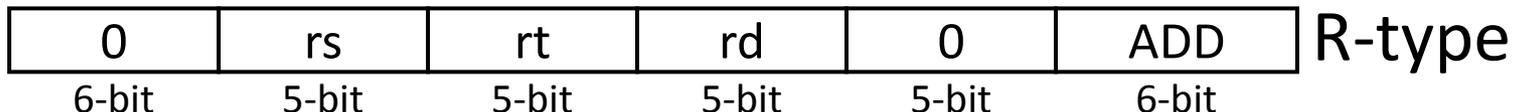
- 4 bytes per instruction, regardless of format
- must be 4-byte aligned (2 lsb of PC must be 2b'00)
- format and fields readily extractable

# ALU Instructions

- ◆ Assembly (e.g., register-register signed addition)

ADD rd<sub>reg</sub> rs<sub>reg</sub> rt<sub>reg</sub>

- ◆ Machine encoding



- ◆ Semantics

- $GPR[rd] \leftarrow GPR[rs] + GPR[rt]$
- $PC \leftarrow PC + 4$

- ◆ Exception on “overflow”

- ◆ Variations

- Arithmetic: {signed, unsigned} x {ADD, SUB}
- Logical: {AND, OR, XOR, NOR}
- Shift: {Left, Right-Logical, Right-Arithmetic}

# Reg-Reg Instruction Encoding

		<b>SPECIAL function</b>							
		2...0							
		0	1	2	3	4	5	6	7
5...3	0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
	1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
	2	MFHI	MTHI	MFLO	MTLO	DSLLV $\epsilon$	*	DSRLV $\epsilon$	DSRAV $\epsilon$
	3	MULT	MULTU	DIV	DIVU	DMULT $\epsilon$	DMULTU $\epsilon$	DDIV $\epsilon$	DDIVU $\epsilon$
	4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
	5	*	*	SLT	SLTU	DADD $\epsilon$	DADDU $\epsilon$	DSUB $\epsilon$	DSUBU $\epsilon$
	6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
	7	DSLL $\epsilon$	*	DSRL $\epsilon$	DSRA $\epsilon$	DSLL32 $\epsilon$	*	DSRL32 $\epsilon$	DSRA32 $\epsilon$

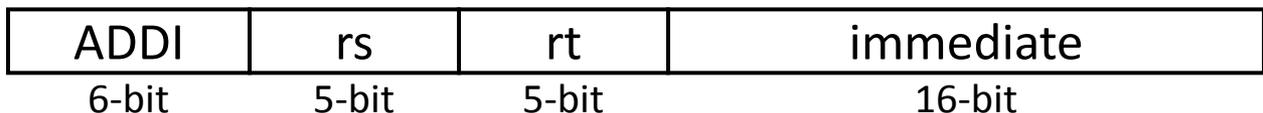
[MIPS R4000 Microprocessor User's Manual]

# ALU Instructions

- ◆ Assembly (e.g., regi-immediate signed additions)

ADDI  $rt_{reg}$   $rs_{reg}$   $immediate_{16}$

- ◆ Machine encoding



I-type

- ◆ Semantics

- $GPR[rt] \leftarrow GPR[rs] + \text{sign-extend}(\text{immediate})$
- $PC \leftarrow PC + 4$

- ◆ Exception on “overflow”

- ◆ Variations

- Arithmetic: {signed, unsigned} x {ADD, ~~SUB~~}
- Logical: {AND, OR, XOR, LUI}

# Reg-Immed Instruction Encoding

		Opcode							
28...26		0	1	2	3	4	5	6	7
31...29	0	SPECIAL	REGIMM	J	JAL	BEQ	BNE	BLEZ	BGTZ
	1	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
	2	COP0	COP1	COP2	*	BEQL	BNEL	BLEZL	BGTZL
	3	DADDI $\epsilon$	DADDIU $\epsilon$	LDL $\epsilon$	LDR $\epsilon$	*	*	*	*
	4	LB	LH	LWL	LW	LBU	LHU	LWR	LWU $\epsilon$
	5	SB	SH	SWL	SW	SDL $\epsilon$	SDR $\epsilon$	SWR	CACHE $\delta$
	6	LL	LWC1	LWC2	*	LLD $\epsilon$	LDC1	LDC2	LD $\epsilon$
	7	SC	SWC1	SWC2	*	SCD $\epsilon$	SDC1	SDC2	SD $\epsilon$

[MIPS R4000 Microprocessor User's Manual]

# Assembly Programming 101

- ◆ Break down high-level program constructs into a sequence of elemental operations

- ◆ E.g. High-level Code

```
f = ( g + h ) - ( i + j )
```

- ◆ Assembly Code

- suppose  $f, g, h, i, j$  are in  $r_f, r_g, r_h, r_i, r_j$
- suppose  $r_{temp}$  is a free register

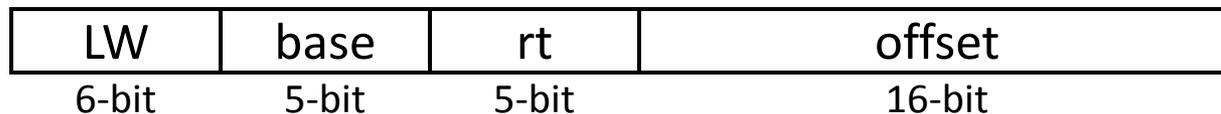
```
add r_temp r_g r_h    # r_temp = g+h
add r_f r_i r_j       # r_f = i+j
sub r_f r_temp r_f    # f = r_temp - r_f
```

# Load Instructions

- ◆ Assembly (e.g., load 4-byte word)

$LW\ rt_{reg}\ offset_{16}\ (base_{reg})$

- ◆ Machine encoding



I-type

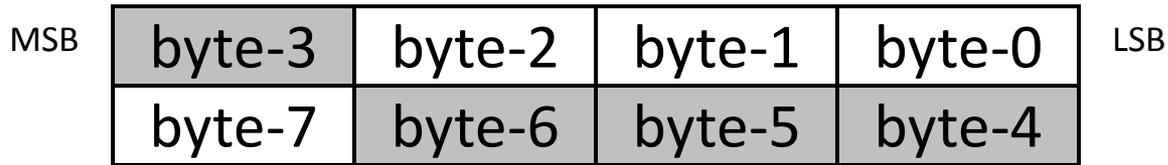
- ◆ Semantics

- $effective\_address = sign\_extend(offset) + GPR[base]$
- $GPR[rt] \leftarrow MEM[ translate(effective\_address) ]$
- $PC \leftarrow PC + 4$

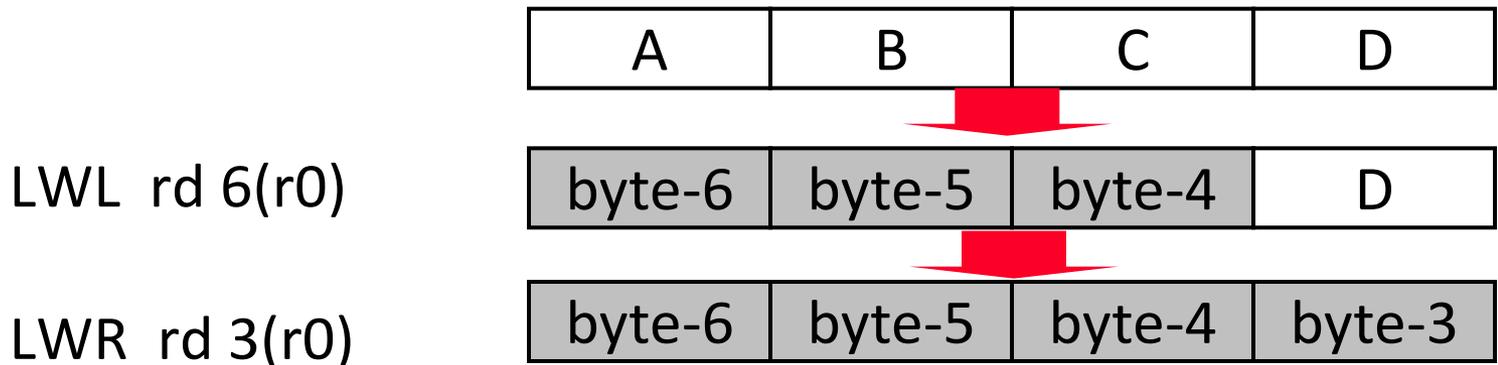
- ◆ Exceptions

- address must be “word-aligned”  
What if you want to load an unaligned word?
- MMU exceptions

# Data Alignment



- ◆ LW/SW alignment restriction
  - not optimized to fetch memory bytes not within a word boundary
  - not optimized to rotate unaligned bytes into registers
- ◆ Provide separate opcodes for the infrequent case



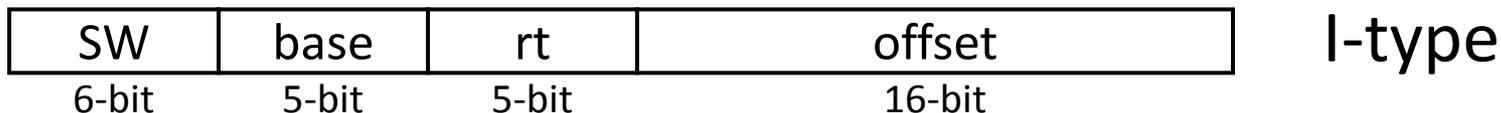
- LWL/LWR is slower but it is okay
- note LWL and LWR still fetch within word boundary

# Store Instructions

- ◆ Assembly (e.g., store 4-byte word)

$SW\ rt_{reg}\ offset_{16}\ (base_{reg})$

- ◆ Machine encoding



- ◆ Semantics

- $effective\_address = sign\_extend(offset) + GPR[base]$
- $MEM[ translate(effective\_address) ] \leftarrow GPR[rt]$
- $PC \leftarrow PC + 4$

- ◆ Exceptions

- address must be “word-aligned”
- MMU exceptions

# Assembly Programming 201

◆ E.g. High-level Code

$$A[8] = h + A[0]$$

where **A** is an array of integers (4-byte each)

◆ Assembly Code

- suppose **&A**, **h** are in  $r_A$ ,  $r_h$
- suppose  $r_{temp}$  is a free register

```
LW  $r_{temp}$  0( $r_A$ )      #  $r_{temp} = A[0]$ 
add  $r_{temp}$   $r_h$   $r_{temp}$  #  $r_{temp} = h + A[0]$ 
SW  $r_{temp}$  32( $r_A$ )    #  $A[8] = r_{temp}$ 
                          # note  $A[8]$  is 32 bytes
                          # from  $A[0]$ 
```

# Control Flow Instructions

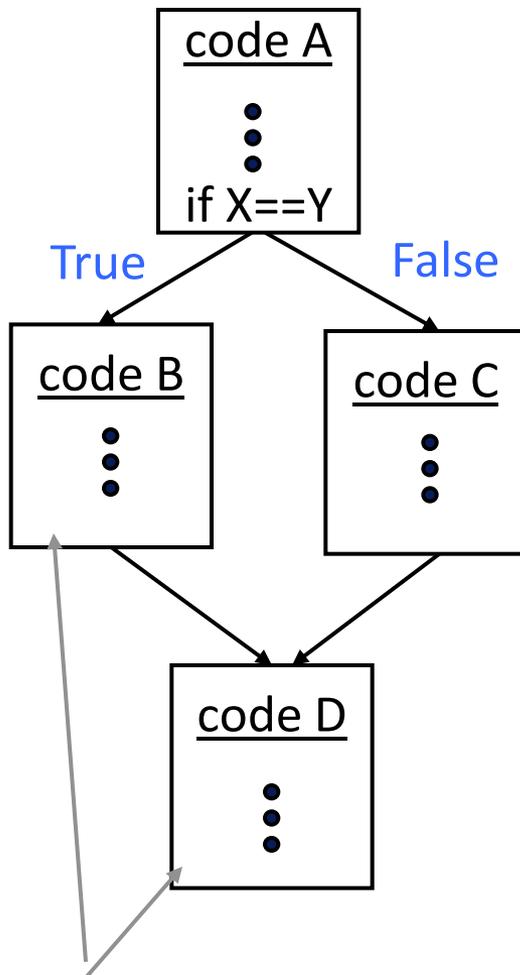
## ◆ C-Code

```

{ code A }
if X==Y then
    { code B }
else
    { code C }
{ code D }

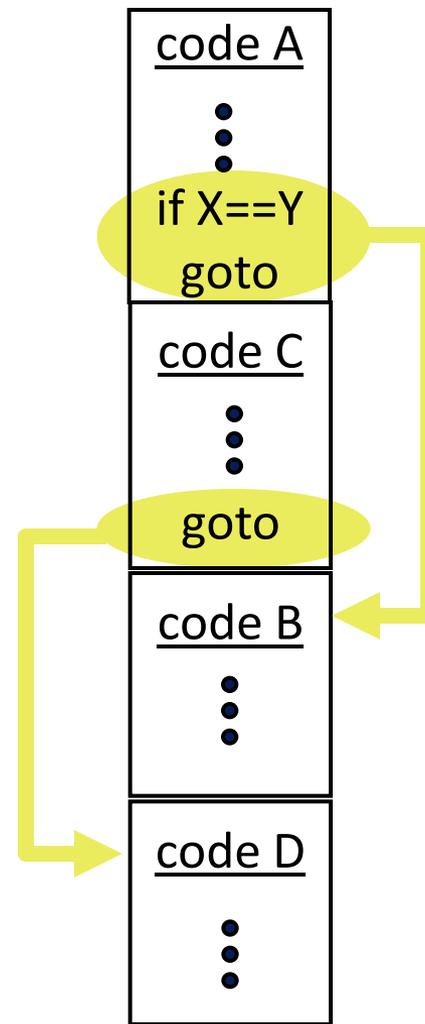
```

## Control Flow Graph



these things are called basic blocks

## Assembly Code (linearized)



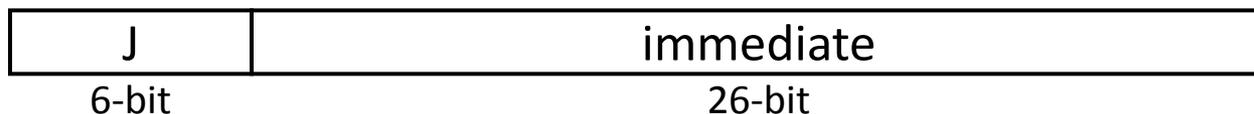


# Jump Instructions

- ◆ Assembly

J immediate<sub>26</sub>

- ◆ Machine encoding



J-type

- ◆ Semantics

- target = PC[31:28]x2<sup>28</sup> | bitwise-or zero-extend(immediate)x4
- PC ← target

- ◆ How far can you jump?

- ◆ Variations

- Jump and Link
- Jump Registers

# Assembly Programming 301

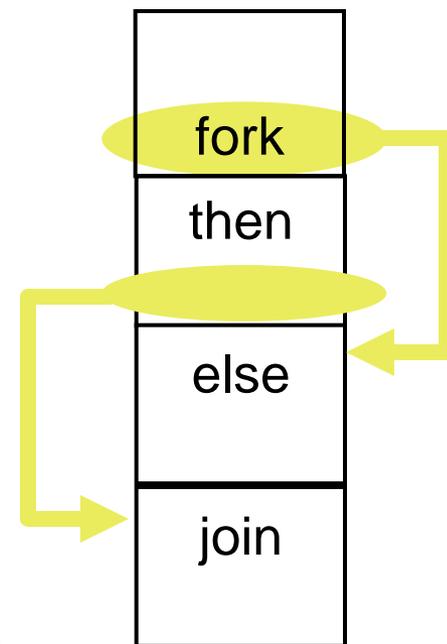
◆ E.g. High-level Code

```

if (i == j) then
    e = g
else
    e = h
f = e
    
```

◆ Assembly Code

- suppose  $e, f, g, h, i, j$  are in  $r_e, r_f, r_g, r_h, r_i, r_j$



```

        bne  r_i  r_j  L1      # L1 and L2 are addr labels
                                # assembler computes offset
        add  r_e  r_g  r0      # e = g
        j   L2
L1:    add  r_e  r_h  r0      # e = h
L2:    add  r_f  r_e  r0      # f = e
        . . . .
    
```

# Function Call and Return

- ◆ Jump and Link:  $JAL\ offset_{26}$ 
  - return address =  $PC + 8$
  - target =  $PC[31:28] \times 2^{28} \mid_{\text{bitwise-or}} \text{zero-extend}(\text{immediate}) \times 4$
  - $PC \leftarrow \text{target}$
  - $GPR[r31] \leftarrow \text{return address}$

On a function call, the callee needs to know where to go back to afterwards

- ◆ Jump Indirect:  $JR\ rs_{\text{reg}}$ 
  - target =  $GPR[rs]$
  - $PC \leftarrow \text{target}$

PC-offset jumps and branches always jump to the same target every time the same instruction is executed

Jump Indirect allows the same instruction to jump to any location specified by  $rs$  (usually  $r31$ )

# Assembly Programming 401

Caller

```
... code A ...
JAL _myfxn
... code C ...
JAL _myfxn
... code D ...
```

Callee

```
_myfxn:    ... code B ...
          JR r31
```

- ◆ ..... **A** →<sub>call</sub> **B** →<sub>return</sub> **C** →<sub>call</sub> **B** →<sub>return</sub> **D** .....
- ◆ How do you pass argument between caller and callee?
- ◆ If **A** set **r10** to **1**, what is the value of **r10** when **B** returns to **C**?
- ◆ What registers can **B** use?
- ◆ What happens to **r31** if **B** calls another function

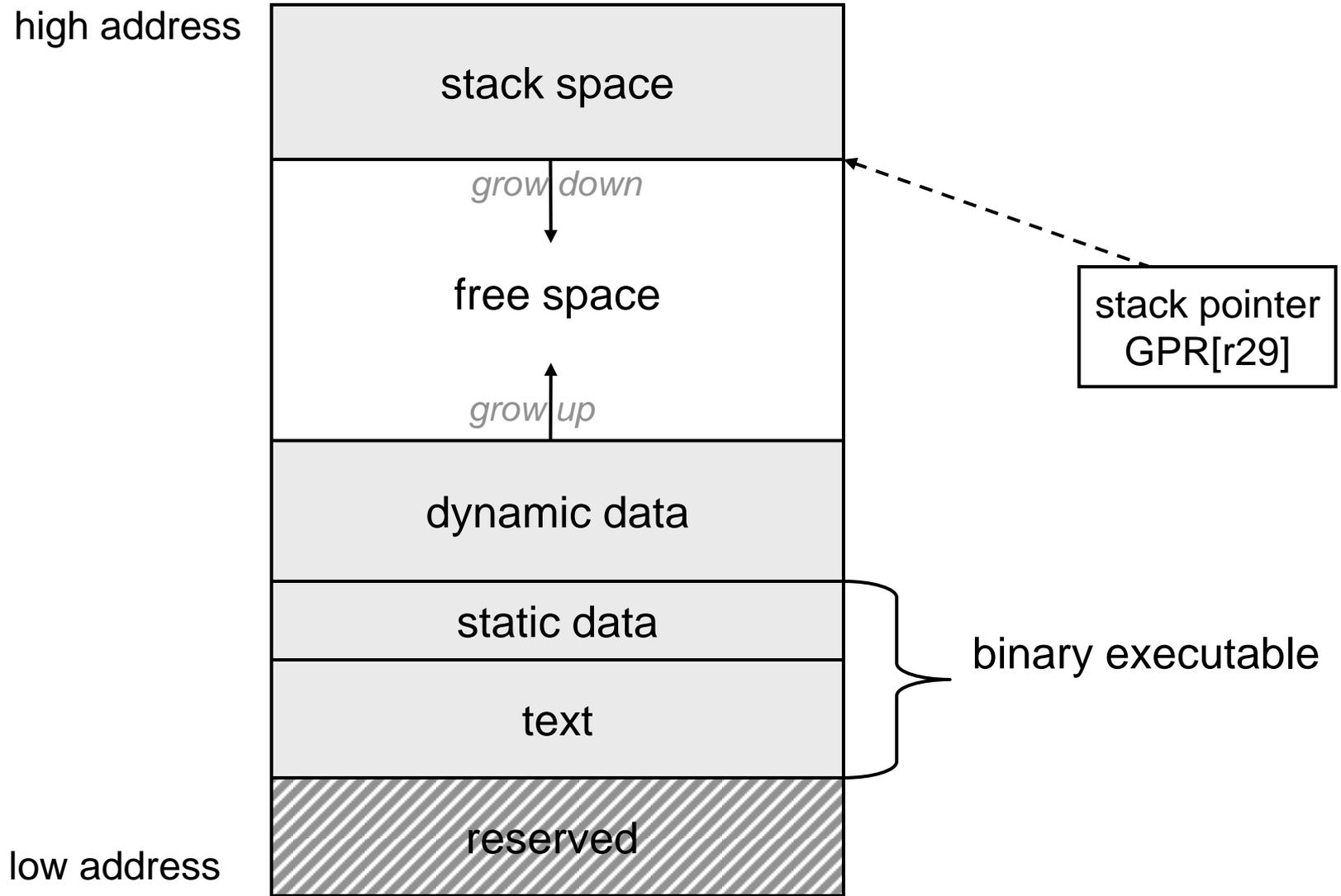
# Caller and Callee Saved Registers

- ◆ Callee-Saved Registers
  - Caller says to callee, “The values of these registers should not change when you return to me.”
  - Callee says, “If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you.”
  
- ◆ Caller-Saved Registers
  - Caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”
  - Callee says to caller, “Don’t count on them staying the same values after I am done.”

# R2000 Register Usage Convention

- ◆ r0: always 0
- ◆ r1: reserved for the assembler
- ◆ r2, r3: function return values
- ◆ r4~r7: function call arguments
- ◆ r8~r15: “caller-saved” temporaries
- ◆ r16~r23 “callee-saved” temporaries
- ◆ r24~r25 “caller-saved” temporaries
- ◆ r26, r27: reserved for the operating system
- ◆ r28: global pointer
- ◆ r29: stack pointer
- ◆ r30: callee-saved temporaries
- ◆ r31: return address

# R2000 Memory Usage Convention



# Calling Convention

- .....
1. caller saves caller-saved registers
  2. caller loads arguments into r4~r7
  3. caller jumps to callee using JAL
  4. callee allocates space on the stack (dec. stack pointer)
  5. callee saves callee-saved registers to stack (also r4~r7, old r29, r31)
- ..... body of callee (can “nest” additional calls) .....
6. callee loads results to r2, r3
  7. callee restores saved register values
  8. JR r31
  9. caller continues with return values in r2, r3
- .....
- prologue
- epilogue

# To Summarize: MIPS RISC

- ◆ Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - few alternatives for accomplishing the same thing
- ◆ Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - “Load-store” architecture
- ◆ Simple branches
  - limited varieties of branch conditions and targets
- ◆ Simple instruction encoding
  - all instructions encoded in the same number of bits
  - only a few formats

Loosely speaking, an ISA intended for compilers rather than assembly programmers

# We didn't talk about

- ◆ Privileged Modes
  - User vs. supervisor
- ◆ Exception Handling
  - trap to supervisor handling routine and back
- ◆ Virtual Memory
  - Each user has 4-GBytes of private, large, linear and fast memory?
- ◆ Floating-Point Instructions

# Load Delay Slots

```

LW    ra    ---
addi  r-    ra    r-
addi  r-    ra    r-
    
```

- ◆ R2000 load has an architectural latency of 1 inst\*.
  - the instruction immediately following a load (in the “delay slot”) still sees the old register value
  - the load instruction no longer has an atomic semantics

Why would you do it this way?

- ◆ Is this a good idea? (hint: R4000 redefined LW to complete atomically)

\*BTW, notice that latency is defined in “instructions” not cyc. or sec.

# Branch Delay Slots

- ◆ R2000 branch instructions also have an architectural latency of 1 instructions
  - the instruction immediately after a branch is always executed (in fact PC-offset is computed from the delay slot instruction)
  - branch target takes effect on the 2<sup>nd</sup> instruction

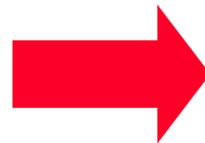
```
bne ri rj L1
```

```
add re rg r0
j L2
```

```
L1: add re rh r0
```

```
L2: add rf re r0
```

```
. . . .
```



```
bne ri rj L1
```

```
nop
```

```
j L2
```

```
add re rg r0
```

```
L1: add re rh r0
```

```
L2: add rf re r0
```

```
. . . .
```