

CMU 18-447 S'13 © 2011 J. C. Hoe

18-447 MIPS ISA

James C. Hoe Dept of ECE, CMU



Instruction Set Architecture

- A stable platform, typically 15~20 years
 - guarantees binary compatibility for SW investments
 - permits adoption of foreseeable technology advances
- User-level ISA
 - program visible state and instructions available to user processes
 - single-user abstraction on top of HW/SW virtualization
- "Virtual Environment" Architecture
 - state and instructions to control virtualization (e.g., caches, sharing)
 - user-level, but not used by your average user programs
- "Operating Environment" Architecture
 - state and instructions to implement virtualization
 - privileged/protected access reserved for OS



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
- How to interface with the outside world?
- Protection and privileged operations
- Software conventions

Very often you compromise immediate optimality for future scalability and compatibility

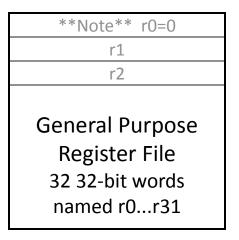


MIPS R2000 Program Visible State

Program Counter

32-bit memory address of the current instruction

M[0]
M[1]
M[2]
M[3]
M[4]
M[N-1]



Memory 2³² by 8-bit locations (4 Giga Bytes) 32-bit address (there is some magic going on)



Data Format

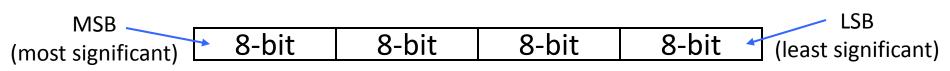
- Most things are 32 bits
 - instruction and data addresses
 - signed and unsigned integers
 - just bits
- Also 16-bit word and 8-bit word (aka byte)
- Floating-point numbers
 - IEEE standard 754
 - float: 8-bit exponent, 23-bit significand
 - double: 11-bit exponent, 52-bit significand



Big Endian vs. Little Endian

(Part I, Chapter 4, Gulliver's Travels)

32-bit signed or unsigned integer comprises 4 bytes



On a byte-addressable machine

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

pointer points to the big end

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

Little Endian

pointer points to the little end

What difference does it make?

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



Instruction Formats

3 simple formats

- R-type, 3 register operands

0	rs	rt	rd	shamt	funct	R-type		
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit			
I-type, 2 register operands and 16-bit immediate								
opcode	rs	rt		l-type				
6-bit	5-bit	5-bit		16-bit				
L-type 2	C hit in			ام ما				

- J-type, 26-bit immediate operand

	opcode	immediate	J-type
-	6-bit	26-bit	

- 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_{reg} rs_{reg} rt_{reg}

Machine encoding

0	rs	rt	rd	0	ADD	R-type
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit	

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

	20		S	PECIAL	function			
53	0	1	2	3	4	5	6	7
0	SLL	*	SRL	SRA	SLLV	*	SRLV	SRAV
1	JR	JALR	*	*	SYSCALL	BREAK	*	SYNC
2	MFHI	MTHI	MFLO	MTLO	DSLLVε	*	DSRLVε	DSRAVε
3	MULT	MULTU	DIV	DIVU	DMULTε	DMULTUε	DDIVε	DDIVUε
4	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	*	*	SLT	SLTU	DADDE	DADDUe	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32e	DSRA32ε

[MIPS R4000 Microprocessor User's Manual]

What patterns do you see? Why are they there?



ALU Instructions

Assembly (e.g., regi-immediate signed additions)
ADDI rt_{reg} rs_{reg} immediate₁₆

Machine encoding

ADDI	rs	rt	immediate	I-type
 6-bit	5-bit	5-bit	16-bit	

Semantics

- GPR[rt] ← GPR[rs] + sign-extend (immediate)
- $PC \leftarrow PC + 4$
- Exception on "overflow"
- Variations
 - Arithmetic: {signed, unsigned} x {ADD, SUB}
 - Logical: {AND, OR, XOR, LUI}



Reg-Immed Instruction Encoding

	2826			Орс	ode			
3129	0	1	2	3	4	5	6	7
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	DADDIE	DADDIUe	LDL	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLe	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDE	SDC1	SDC2	SDε

[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
$$\mathbf{r}_{\text{temp}} \mathbf{r}_{\text{g}} \mathbf{r}_{\text{h}}$$
 # $\mathbf{r}_{\text{temp}} = g+h$
add $\mathbf{r}_{\text{f}} \mathbf{r}_{\text{i}} \mathbf{r}_{\text{j}}$ # $\mathbf{r}_{\text{f}} = i+j$
sub $\mathbf{r}_{\text{f}} \mathbf{r}_{\text{temp}} \mathbf{r}_{\text{f}}$ # $\mathbf{f} = \mathbf{r}_{\text{temp}} - \mathbf{r}_{\text{f}}$



Load Instructions

Assembly (e.g., load 4-byte word)

LW rt_{reg} offset₁₆ (base_{reg})

Machine encoding

LW	base	rt	offset	l-type
6-bit	5-bit	5-bit	16-bit	

Semantics

- effective_address = sign-extend(offset) + GPR[base]
- GPR[rt] ← 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

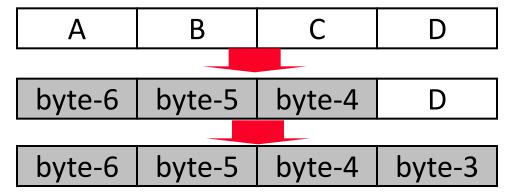
MSB	byte-3 byte-2		byte-1	byte-0	LSB
	byte-7	byte-6	byte-5	byte-4	

LW/SW alignment restriction

LWL rd 6(r0)

LWR rd 3(r0)

- 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

CMU 18-447 S'13 © 2011 J. C. Hoe



Store Instructions

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

SW rt_{reg} offset₁₆ (base_{reg})

Machine encoding

SW	base	rt	offset	l-type
6-bit	5-bit	5-bit	16-bit	

Semantics

- effective_address = sign-extend(offset) + GPR[base]
- MEM[translate(effective_address)] ← GPR[rt]
- $PC \leftarrow PC + 4$
- Exceptions
 - address must be "word-aligned"
 - MMU exceptions



Assembly Programming 201

E.g. High-level Code

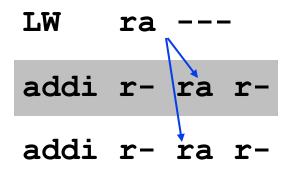
$$A[8] = h + A[0]$$

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



Load Delay Slots



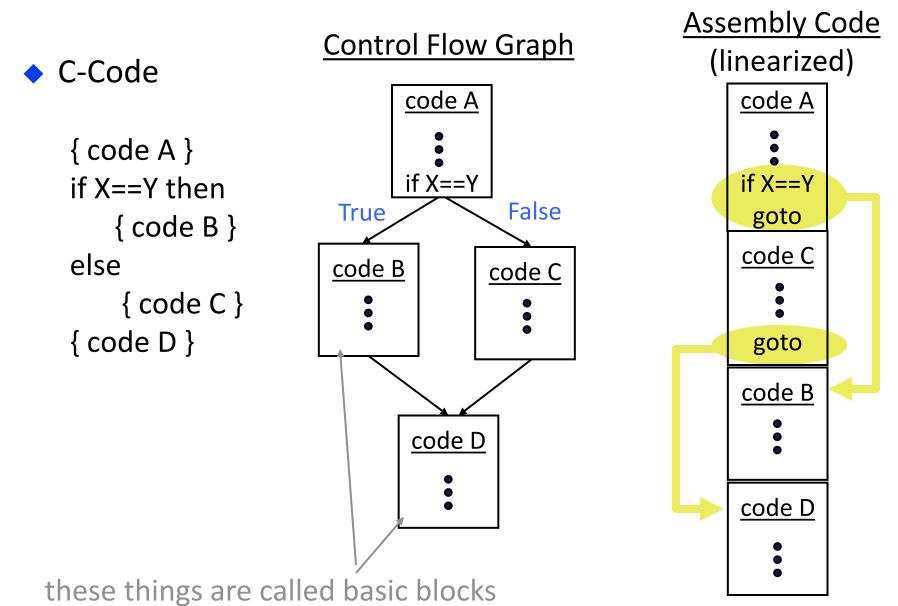
R2000 load has an architectural latency of 1 inst*.

- 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 <u>redefined</u> LW to complete atomically)

*BTW, notice that latency is defined in "instructions" not cyc. or sec.



Control Flow Instructions





(Conditional) Branch Instructions

Assembly (e.g., branch if equal)
BEQ rs_{reg} rt_{reg} immediate₁₆

Machine encoding

BEQ	rs	rt	immediate	l-type
6-bit	5-bit	5-bit	16-bit	

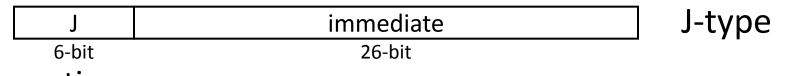
Semantics

- target = PC + sign-extend(immediate) x 4
- if GPR[rs]==GPR[rt] then $PC \leftarrow target$
 - else $PC \leftarrow PC + 4$
- How far can you jump?
- Variations
 - BEQ, BNE, BLEZ, BGTZ



Jump Instructions

- Assembly
 - J immediate₂₆
- Machine encoding

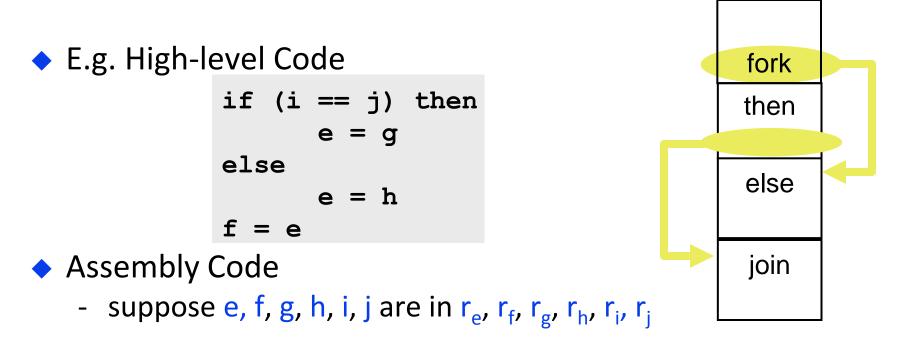


Semantics

- target = PC[31:28]x2²⁸ |_{bitwise-or} zeroextend(immediate)x4
- PC \leftarrow target
- How far can you jump?
- Variations
 - Jump and Link
 - Jump Registers



Assembly Programming 301

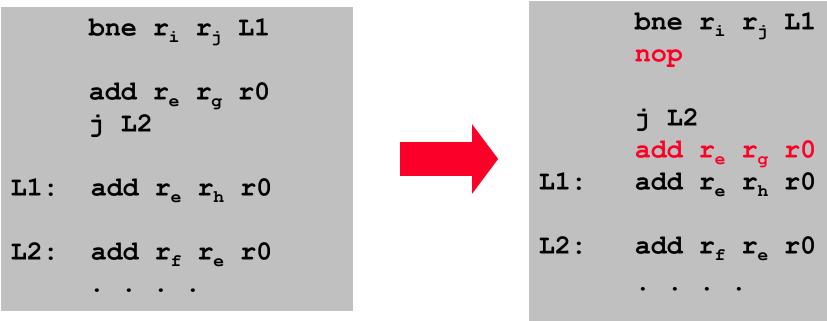


	bne r _i r _j L1	#	L	L a	and L2	are	addr	labels
		#	a	sse	embler	comp	outes	offset
	add r _e r _g r0	#	е	=	g			
	j L2							
L1:	add r _e r _h r0	#	е	=	h			
L2 :	add r _f r _e r0	#	f	=	е			



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 2nd instruction





Strangeness in the Semantics

Where do you think you will end up?

_s:	j L1 j L2 j L3
L1:	j L4
L2:	j L5
L3:	foo
L4:	bar
L5:	baz



Function Call and Return

- Jump and Link: JAL offset₂₆
 - return address = PC + 8
 - target = PC[31:28]x2²⁸ |_{bitwise-or} zeroextend(immediate)x4
 - $PC \leftarrow target$
 - GPR[r31] ← return address

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

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

<u>Caller</u>	<u>Callee</u>	
code A	_myfxn:	code B
JAL _myfxn		JR r31
code C		
JAL _myfxn		
code D		

- $\diamond \dots A \rightarrow_{\mathsf{call}} B \rightarrow_{\mathsf{return}} C \rightarrow_{\mathsf{call}} B \rightarrow_{\mathsf{return}} D \dots$
- 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.

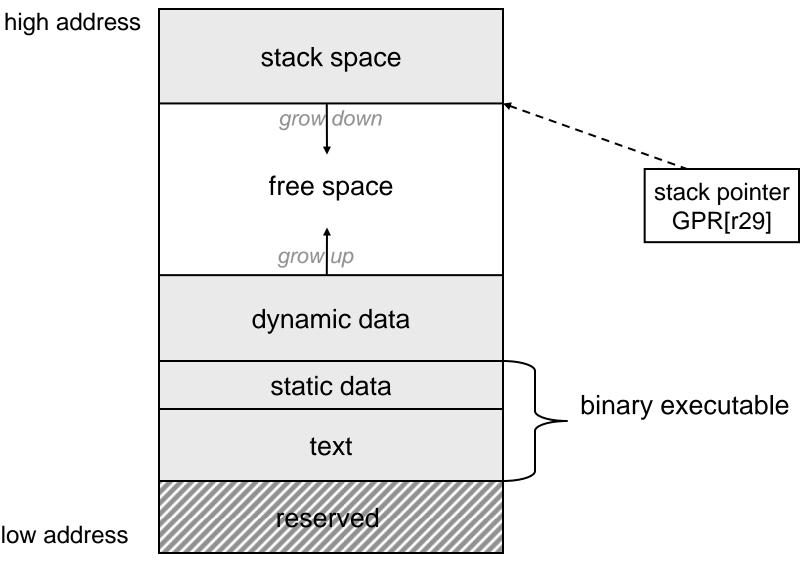


R2000 Register Usage Convention

- 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



CMU 18-447 S'13 © 2011 J. C. Hoe



low address



.

prologue

pilogu

Calling Convention

- 1. caller saves caller-saved registers
- 2. caller loads arguments into r4~r7
- 3. caller jumps to callee using JAL
- callee allocates space on the stack (dec. stack pointer)
- 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

.



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