Lecture 3: Instruction Set Architecture (ISA) part.2 (MIPS)

Jangwoo Kim (Seoul National University)
jangwoo@snu.ac.kr

Announcement

Announcements:	*HW #1 will be posted on eTL site soon.*
Reading:	*Start reading P&H Ch.3 (stuff for the next lectures).*
Handouts:	*None*

[Important] Auditing Policy

- **Too many "audit" requests!!**
  - Students from the other session
  - Students interested in doing lab only
  - Students wanting to study in advance
  - Graduate students in other fields

Basic Policy:

(1) If not registered nor audited, DO NOT attend lecture.

**(2) Audit will be allowed only for graduate students
(or some students who I specifically allow)**

(3) Audit students should not take exam nor do lab

Review

- Architecture = programmer visible state
- What is "programmer visible state?"
  - Who is "programmer" in this context?
  - This is often called as "architectural state"

Useful thinking:

- Writing a system program. What should we know?
  - Operating system, architecture simulator?
- Writing a compiler. What should we know?
  - C code → ?
- Writing an OS for context switching. What should we know?

  context-out: a program running in CPU → ?

  context-in: ? → an empty CPU

Instruction Set Architecture

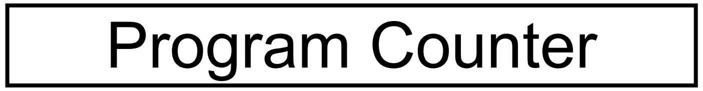
- A stable platform, typically 15~20 years
  - Guarantees binary compatibility for SW investments
  - Permits adoption of foreseeable technology advances
- User-level ISA
  - Program visible state and instructions available to user processes
  - Single-user abstraction on top of HW/SW virtualization
- "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

What are specified/decided in an ISA?

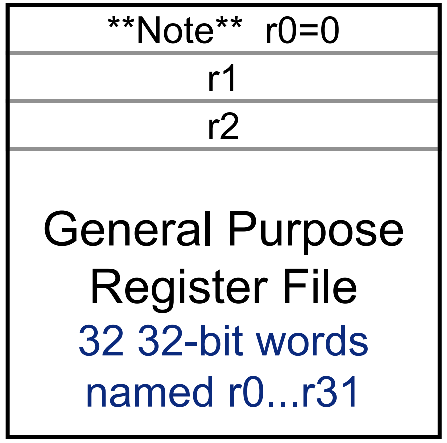
- Data format and size
  - character, binary, decimal, floating point, negatives
- "Programmer Visible State"
  - memory, registers, program counters, etc.
- Instructions: how to change the programmer visible state?
  - what to perform and what to perform next
  - where are the operands
- Instruction-to-binary 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

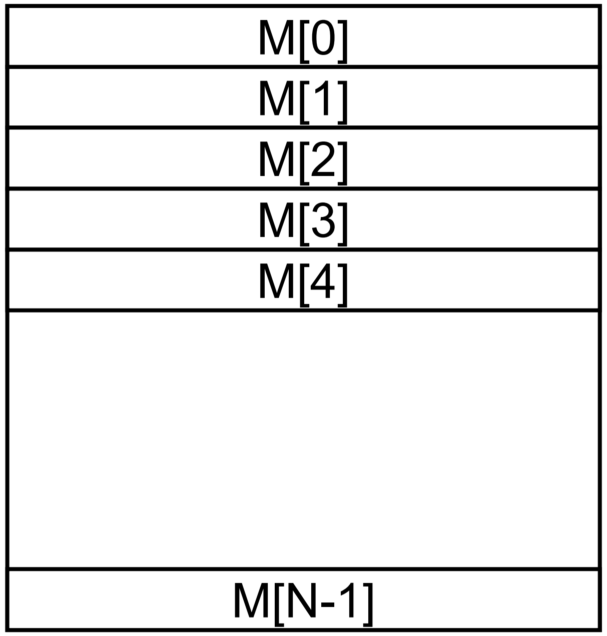
MIPS R2000 Program Visible State



32-bit memory address
of the current instruction



32 32-bit words
named r0...r31



Memory

2^{32} by 8-bit locations (4 Giga Bytes)

32-bit address

Data Format

- Most things are 32 bits (= word)
  - Instruction and data addresses
  - Signed and unsigned integers
  - Just bits
- Also 16-bit word and 8-bit word (= byte)
- Floating-point numbers
  - IEEE standard 754
  - Single precision:

    1-bit sign (S), 8-bit exponent (E), 23-bit fraction (F)
    - $(-1)^S \times F \times 2^{(\text{exponent})}$ ← Not exactly this though.
    - Fraction (aka significand)

    e.g., $(-1)^1 \times (11) \times 2^{-(10)}$ =?
  - Double precision:

    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

MSB (most significant)					LSB (least significant)
	8-bit	8-bit	8-bit	8-bit	

- On a byte-addressable machine . . . .

Big Endian

MSB Addr $\longrightarrow$ 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 <u>big end</u>

Little Endian

LSB Addr $\longrightarrow$ MSB

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 <u>little end</u>

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
6-bit	5-bit	5-bit	5-bit	5-bit	6-bit

R-type

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

opcode	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

  - J-type, 26-bit immediate operand

opcode	immediate
6-bit	26-bit

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)

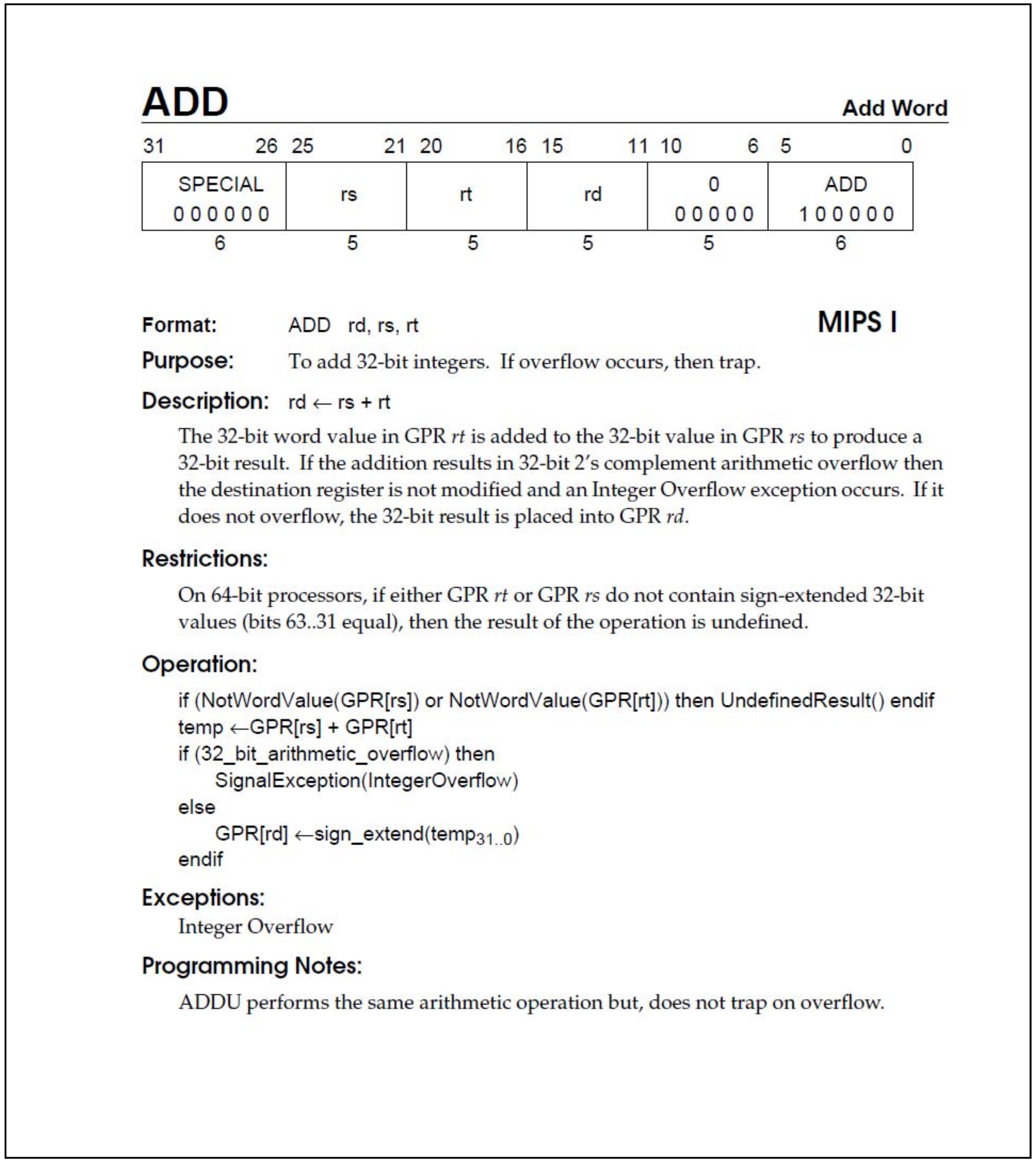
  ADD rd_{reg} rs_{reg} rt_{reg}

- Machine encoding

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

R-type

- Semantics **// how arch. state is changed**
  - 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}



ADD

Add Word

31 26	25 21	20 16	15 11	10 6	5 0
SPECIAL 000000	rs	rt	rd	0 00000	ADD 100000
6	5	5	5	5	6

Format: ADD rd, rs, rt **MIPS I**

Purpose: To add 32-bit integers. If overflow occurs, then trap.

Description: rd ← rs + rt

The 32-bit word value in GPR *rt* is added to the 32-bit value in GPR *rs* to produce a 32-bit result. If the addition results in 32-bit 2's complement arithmetic overflow then the destination register is not modified and an Integer Overflow exception occurs. If it does not overflow, the 32-bit result is placed into GPR *rd*.

Restrictions:

On 64-bit processors, if either GPR *rt* or GPR *rs* do not contain sign-extended 32-bit values (bits 63..31 equal), then the result of the operation is undefined.

Operation:

```
if (NotWordValue(GPR[rs]) or NotWordValue(GPR[rt])) then UndefinedResult() endif
temp ←GPR[rs] + GPR[rt]
if (32_bit_arithmetic_overflow) then
    SignalException(IntegerOverflow)
else
    GPR[rd] ←sign_extend($temp_{31..0}$)
endif
```

Exceptions:

Integer Overflow

Programming Notes:

ADDU performs the same arithmetic operation but, does not trap on overflow.

From the ISA manual

Reg-Reg Instruction Encoding

SPECIAL function

5...3 \ 2...0	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	DADDε	DADDUε	DSUBε	DSUBUε
6	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	DSLLε	*	DSRLε	DSRAε	DSLL32ε	*	DSRL32ε	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)

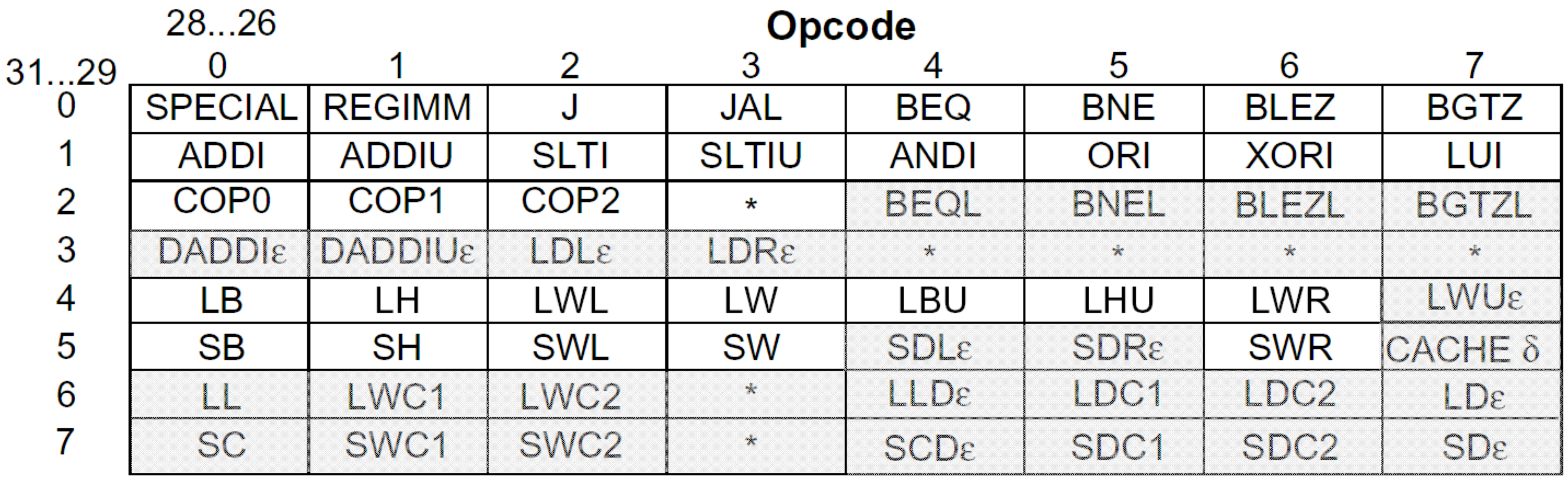
  ADDI rt_{reg} rs_{reg} $immediate_{16}$
- Machine encoding

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

I-type

- Semantics
  - GPR[rt] $\leftarrow$ 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



Opcode

31...29 \ 28...26	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	DADDIε	DADDIUε	LDLε	LDRε	*	*	*	*
4	LB	LH	LWL	LW	LBU	LHU	LWR	LWUε
5	SB	SH	SWL	SW	SDLε	SDRε	SWR	CACHE δ
6	LL	LWC1	LWC2	*	LLDε	LDC1	LDC2	LDε
7	SC	SWC1	SWC2	*	SCDε	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 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)

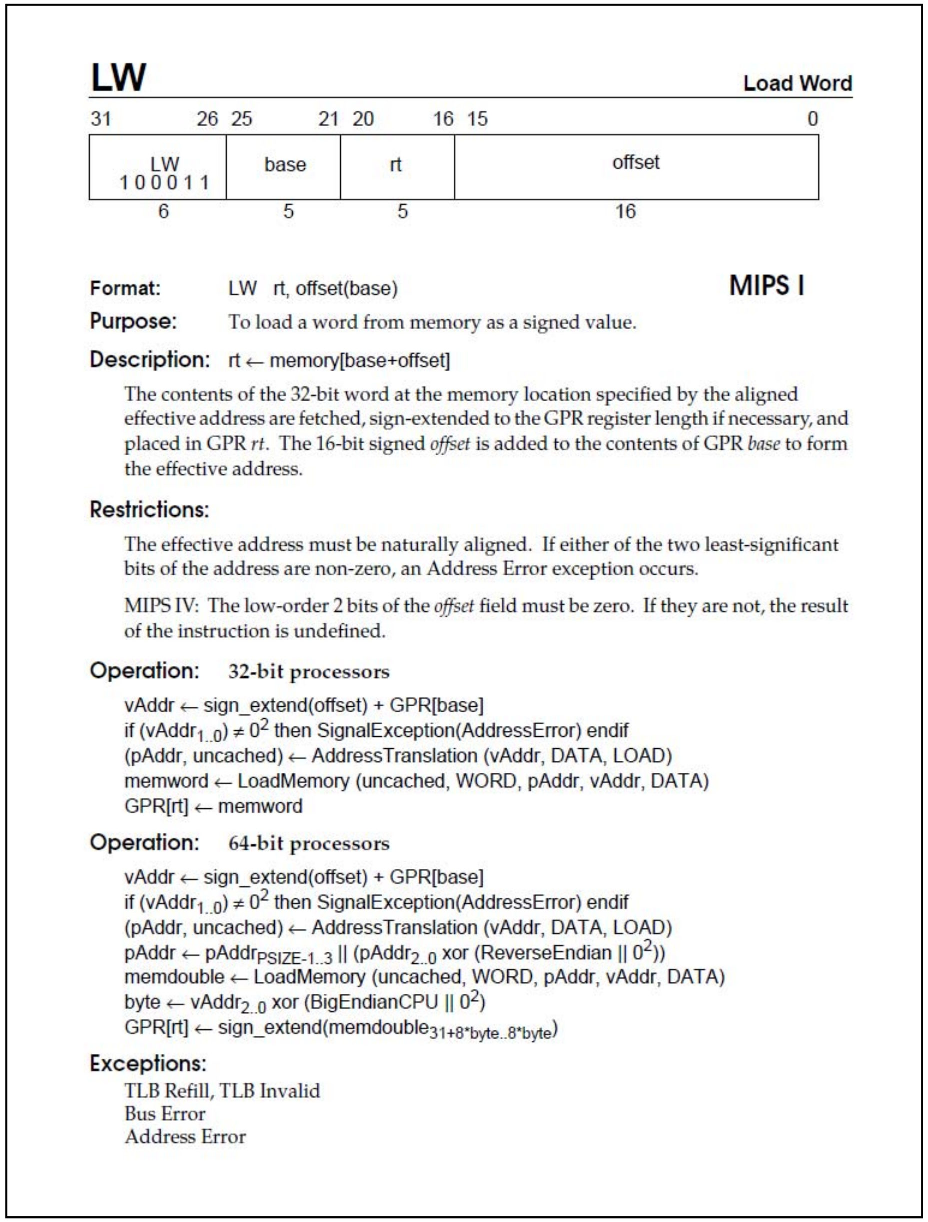
  LW rt_{reg} $offset_{16}$ ($base_{reg}$)
- Machine encoding

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

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 $\leftarrow$ Let's handle unexpected behaviors.



LW — Load Word

31–26	25–21	20–16	15–0
LW 100011	base	rt	offset
6	5	5	16

Format: LW rt, offset(base) **MIPS I**

Purpose: To load a word from memory as a signed value.

Description: rt ← memory[base+offset]

The contents of the 32-bit word at the memory location specified by the aligned effective address are fetched, sign-extended to the GPR register length if necessary, and placed in GPR *rt*. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
memword ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
GPR[rt] ← memword
```

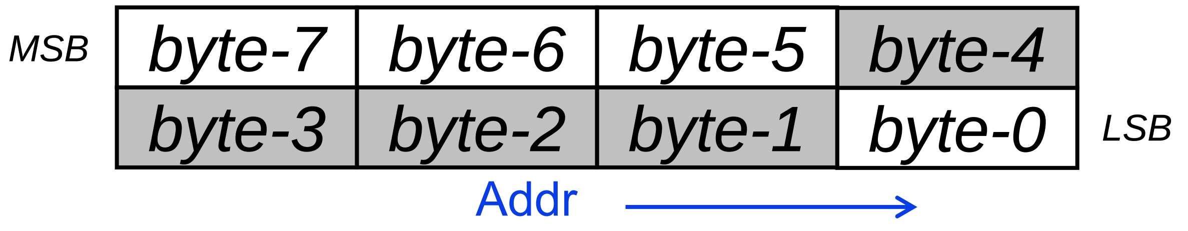
Operation: 64-bit processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, LOAD)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$))
memdouble ← LoadMemory (uncached, WORD, pAddr, vAddr, DATA)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
GPR[rt] ← sign_extend($memdouble_{31+8*byte..8*byte}$)
```

Exceptions:

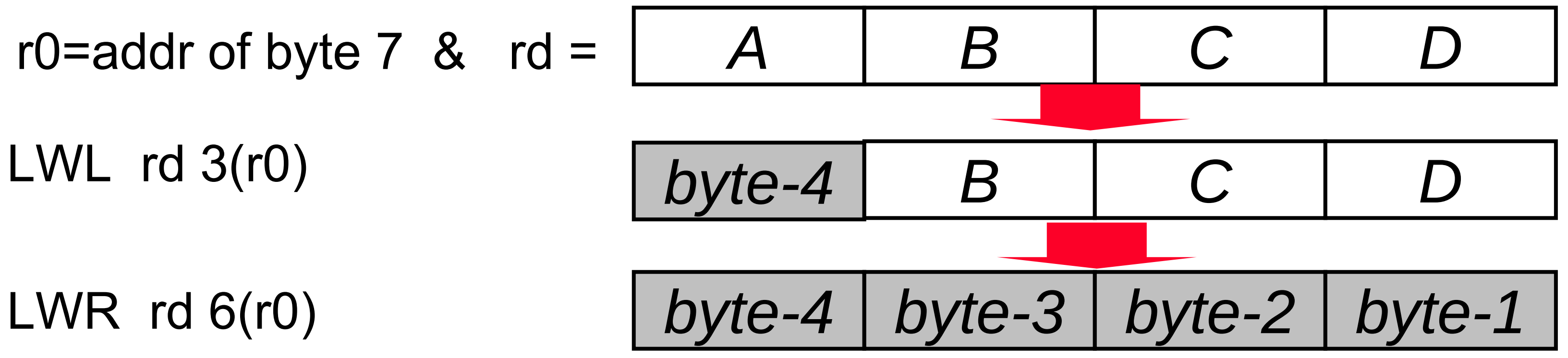
TLB Refill, TLB Invalid
Bus Error
Address Error

Data Alignment



Big Endian

- LW/SW alignment restriction
  - Not optimized to fetch memory bytes not within a word boundary
  - Not optimized to rotate unaligned bytes into registers
- Can use 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)

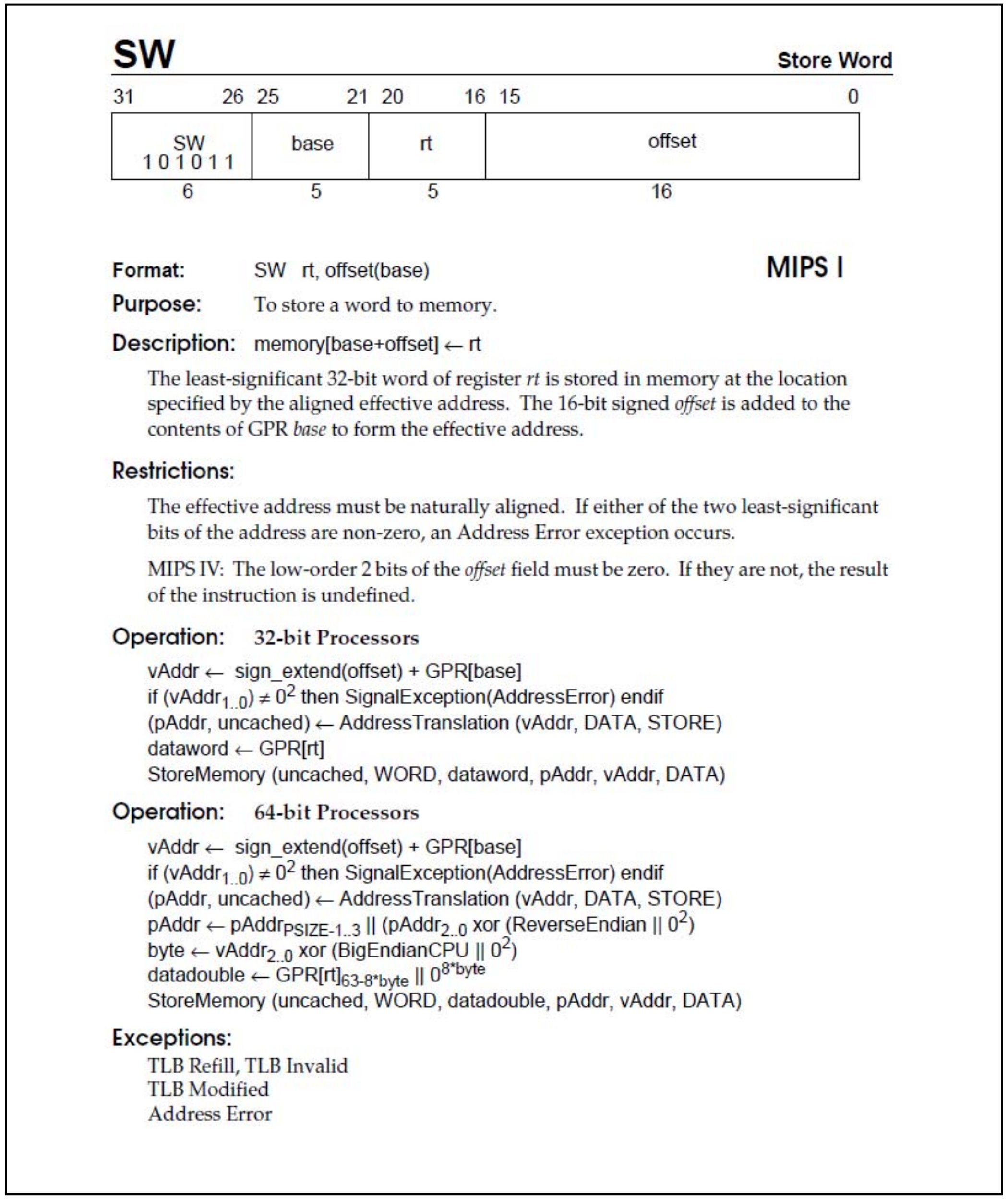
  SW $\mathrm{rt}_{\mathrm{reg}}$ offset_{16} $(\mathrm{base}_{\mathrm{reg}})$

- Machine encoding

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

I-type

- 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



SW

Store Word

31–26	25–21	20–16	15–0
SW 101011	base	rt	offset
6	5	5	16

Format: SW rt, offset(base) **MIPS I**

Purpose: To store a word to memory.

Description: memory[base+offset] ← rt

The least-significant 32-bit word of register *rt* is stored in memory at the location specified by the aligned effective address. The 16-bit signed *offset* is added to the contents of GPR *base* to form the effective address.

Restrictions:

The effective address must be naturally aligned. If either of the two least-significant bits of the address are non-zero, an Address Error exception occurs.

MIPS IV: The low-order 2 bits of the *offset* field must be zero. If they are not, the result of the instruction is undefined.

Operation: 32-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
dataword ← GPR[rt]
StoreMemory (uncached, WORD, dataword, pAddr, vAddr, DATA)
```

Operation: 64-bit Processors

```
vAddr ← sign_extend(offset) + GPR[base]
if ($vAddr_{1..0}$) ≠ $0^2$ then SignalException(AddressError) endif
(pAddr, uncached) ← AddressTranslation (vAddr, DATA, STORE)
pAddr ← $pAddr_{PSIZE-1..3}$ || ($pAddr_{2..0}$ xor (ReverseEndian || $0^2$)
byte ← $vAddr_{2..0}$ xor (BigEndianCPU || $0^2$)
datadouble ← $GPR[rt]_{63-8*byte}$ || $0^{8*byte}$
StoreMemory (uncached, WORD, datadouble, pAddr, vAddr, DATA)
```

Exceptions:

TLB Refill, TLB Invalid
TLB Modified
Address Error

Assembly Programming 201

- E.g. High-level Code

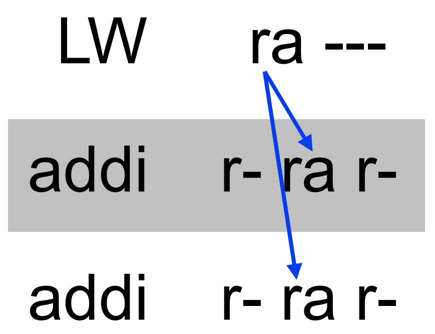
```
A[ 8 ] = h + A[ 0 ]
```

  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]
```

Load Delay Slots (old days..)



```
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?

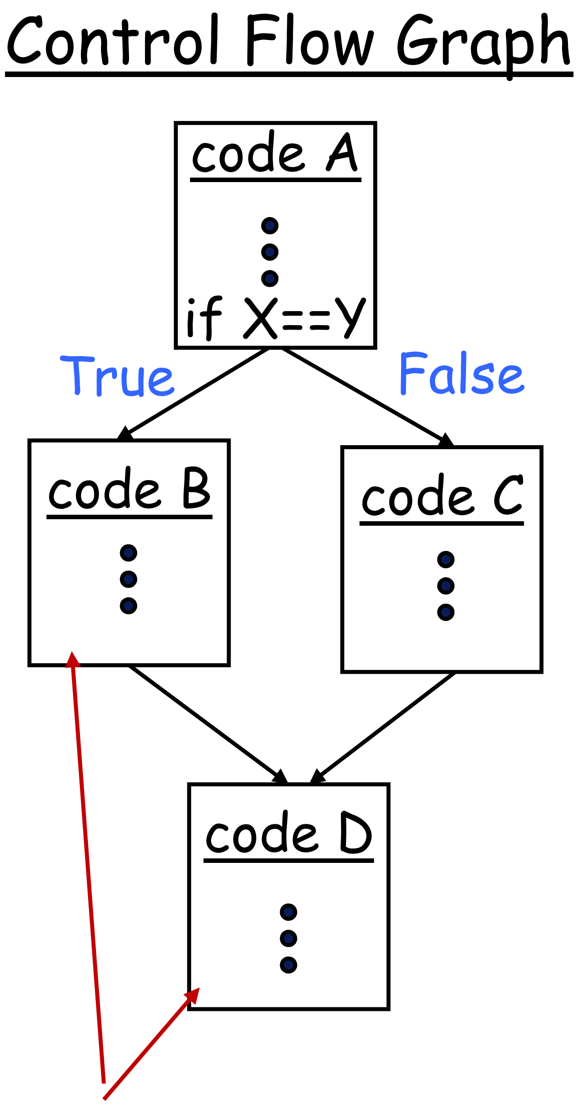
(hint: R4000 <u>redefined</u> LW to complete atomically)

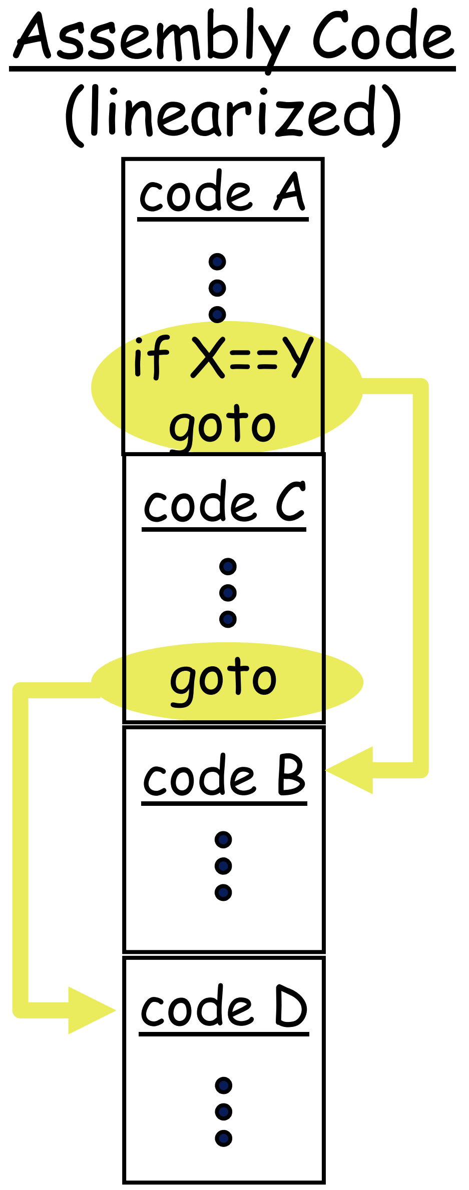
*BTW, notice the latency is defined in "instructions" not cyc. or sec.

Control Flow Instructions

- C-Code

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





these things are called basic blocks

(Conditional) Branch Instructions

- Assembly (e.g., branch if equal)

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

- Machine encoding

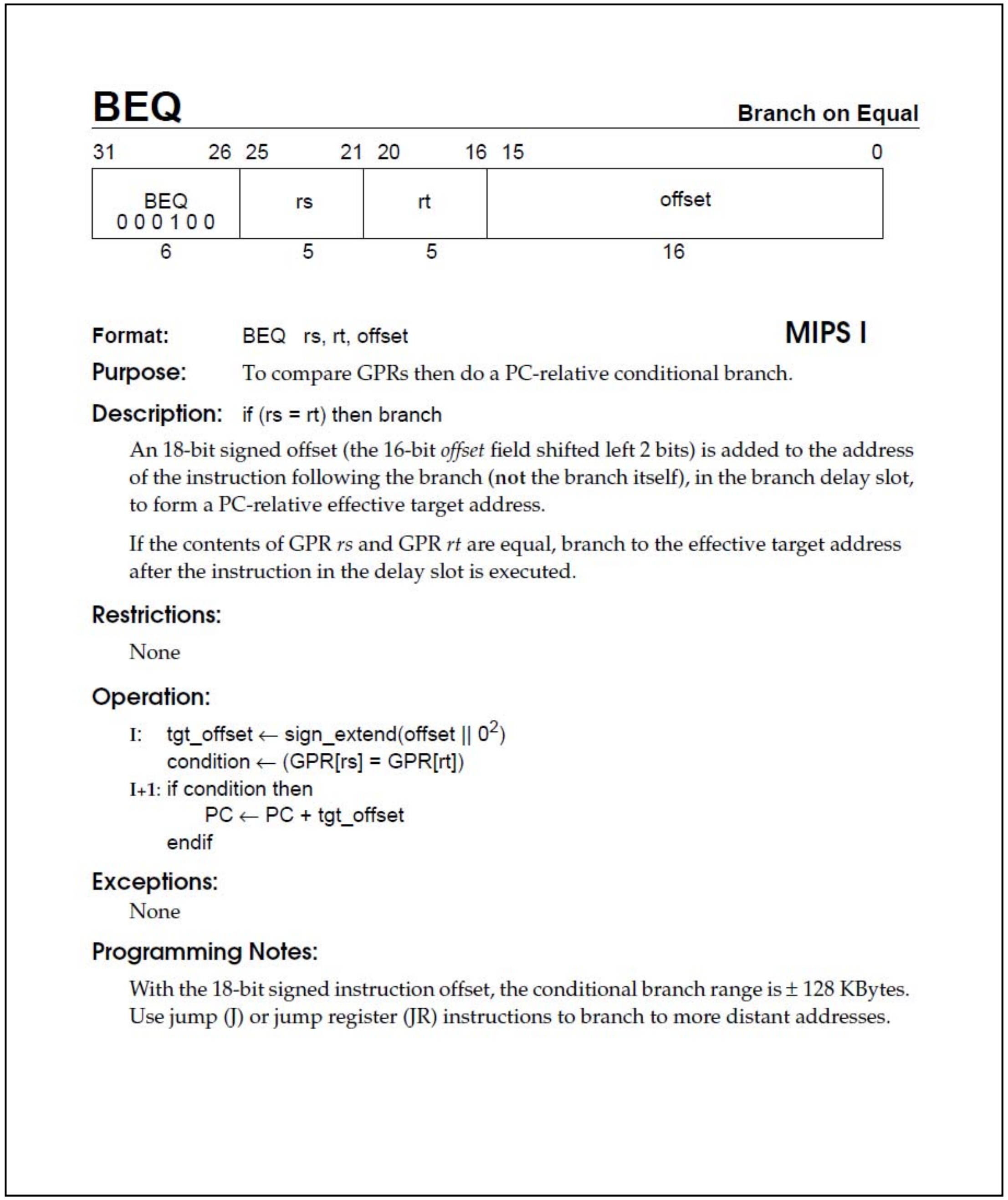
BEQ	rs	rt	immediate
6-bit	5-bit	5-bit	16-bit

I-type

- Semantics // PC-relative jump
  - target = PC + sign-extend(immediate) x 4 ← (x 4) = (<< 2b)
  - if GPR[rs]==GPR[rt] then PC ← target

    else PC ← PC + 4
- How far can you jump?
- Variations
  - BEQ, BNE, BLEZ, BGTZ

Why isn't there a BLT or BGT instruction?



BEQ

Branch on Equal

31 26	25 21	20 16	15 0
BEQ 0 0 0 1 0 0	rs	rt	offset
6	5	5	16

Format: BEQ rs, rt, offset **MIPS I**

Purpose: To compare GPRs then do a PC-relative conditional branch.

Description: if (rs = rt) then branch

An 18-bit signed offset (the 16-bit *offset* field shifted left 2 bits) is added to the address of the instruction following the branch (**not** the branch itself), in the branch delay slot, to form a PC-relative effective target address.

If the contents of GPR *rs* and GPR *rt* are equal, branch to the effective target address after the instruction in the delay slot is executed.

Restrictions:

None

Operation:

```
I:    tgt_offset ← sign_extend(offset || $0^2$)
      condition ← (GPR[rs] = GPR[rt])
I+1:  if condition then
          PC ← PC + tgt_offset
      endif
```

Exceptions:

None

Programming Notes:

With the 18-bit signed instruction offset, the conditional branch range is ± 128 KBytes. Use jump (J) or jump register (JR) instructions to branch to more distant addresses.

Jump Instructions

- Assembly

  J immediate_{26}

- Machine encoding

J	immediate
6-bit	26-bit

J-type

- Semantics // PC-relative jump
  - target = PC[31:28]x2^{28} $|_{\text{bitwise-or}}$ zero-extend(immediate) x 4
  - PC $\leftarrow$ target
- How far can you jump (in both direction)?
- Variations
  - Jump and Link
  - Jump Registers

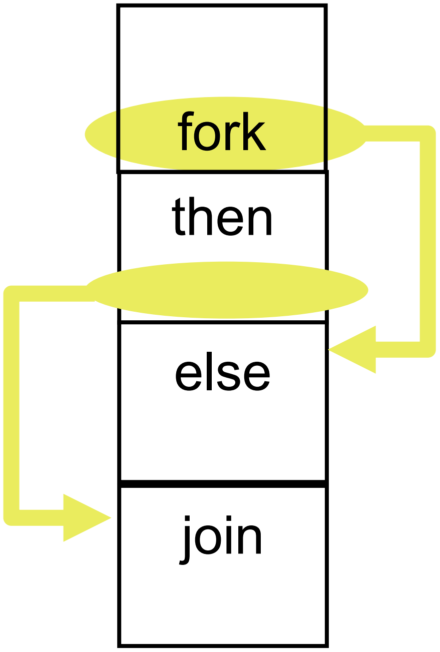
Assembly Programming 301

- 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 ri rj L1         # L1 and L2 are addr labels
                            # assembler computes offset
       add re rg r0         # e = g
       j L2
L1:    add re rh r0         # e = h
L2:    add rf re r0         # f = e
       . . . .
```



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

```
        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
        . . . .
```

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
```

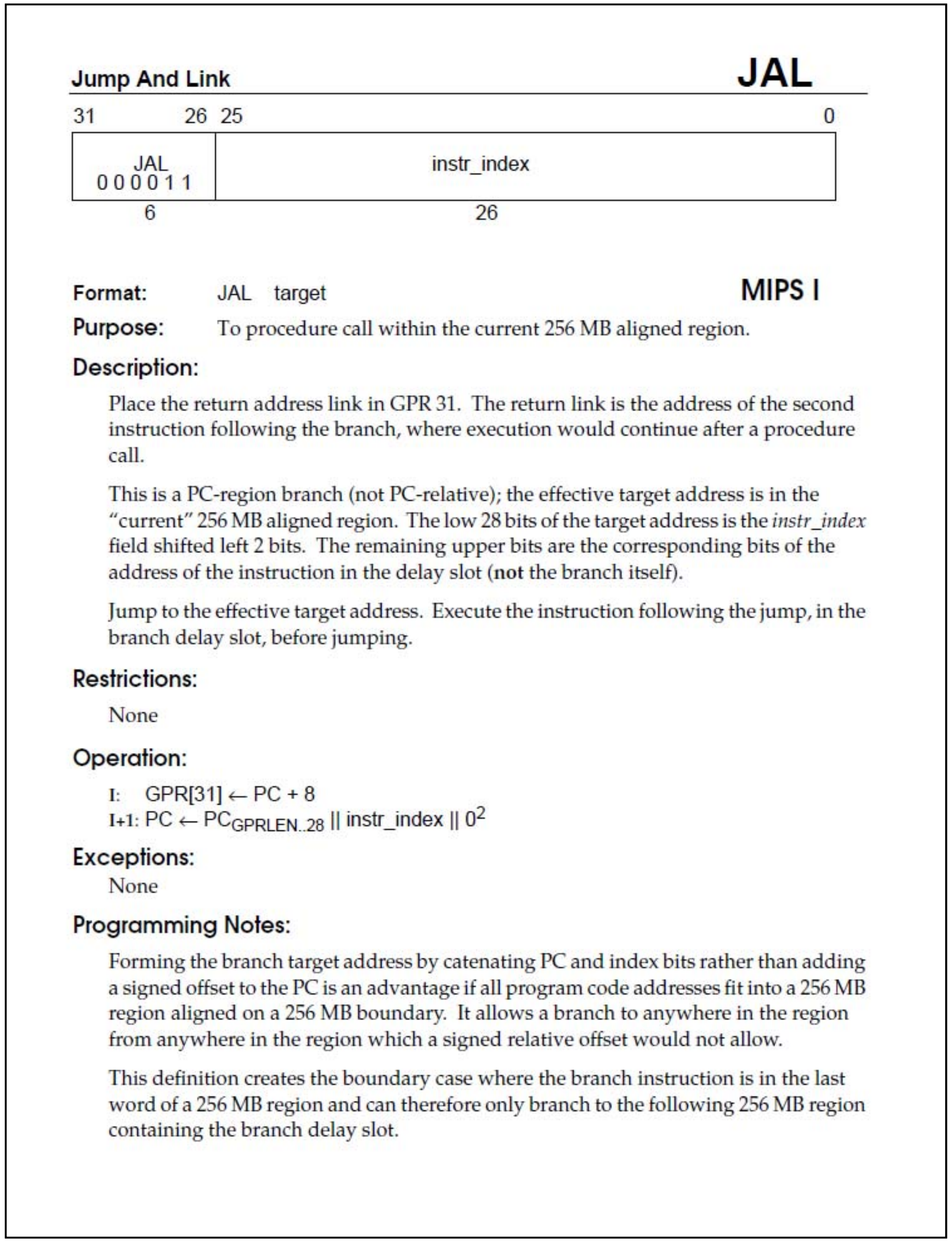
Function Call and Return

- Jump and Link: JAL offset_{26}
  - return address = PC + 8
  - target = $\text{PC}[31{:}28]\text{x}2^{28}\ |_{\text{bitwise-or}}$ zero-extend(immediate)x4
  - PC $\leftarrow$ target
  - GPR[r31] $\leftarrow$ 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)



Jump And Link **JAL**

31 26	25 0
JAL 000011	instr_index
6	26

Format: JAL target **MIPS I**

Purpose: To procedure call within the current 256 MB aligned region.

Description:

Place the return address link in GPR 31. The return link is the address of the second instruction following the branch, where execution would continue after a procedure call.

This is a PC-region branch (not PC-relative); the effective target address is in the "current" 256 MB aligned region. The low 28 bits of the target address is the *instr_index* field shifted left 2 bits. The remaining upper bits are the corresponding bits of the address of the instruction in the delay slot (**not** the branch itself).

Jump to the effective target address. Execute the instruction following the jump, in the branch delay slot, before jumping.

Restrictions:

None

Operation:

I: $GPR[31] \leftarrow PC + 8$
I+1: $PC \leftarrow PC_{GPRLEN..28} \;||\; instr\_index \;||\; 0^2$

Exceptions:

None

Programming Notes:

Forming the branch target address by catenating PC and index bits rather than adding a signed offset to the PC is an advantage if all program code addresses fit into a 256 MB region aligned on a 256 MB boundary. It allows a branch to anywhere in the region from anywhere in the region which a signed relative offset would not allow.

This definition creates the boundary case where the branch instruction is in the last word of a 256 MB region and can therefore only branch to the following 256 MB region containing the branch delay slot.

Assembly Programming 401

Caller

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

Callee

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

- ..... **A** $\rightarrow_{\text{call}}$ **B** $\rightarrow_{\text{return}}$ **C** $\rightarrow_{\text{call}}$ **B** $\rightarrow_{\text{return}}$ **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."

  → "hey caller! I am giving you back original values on return."

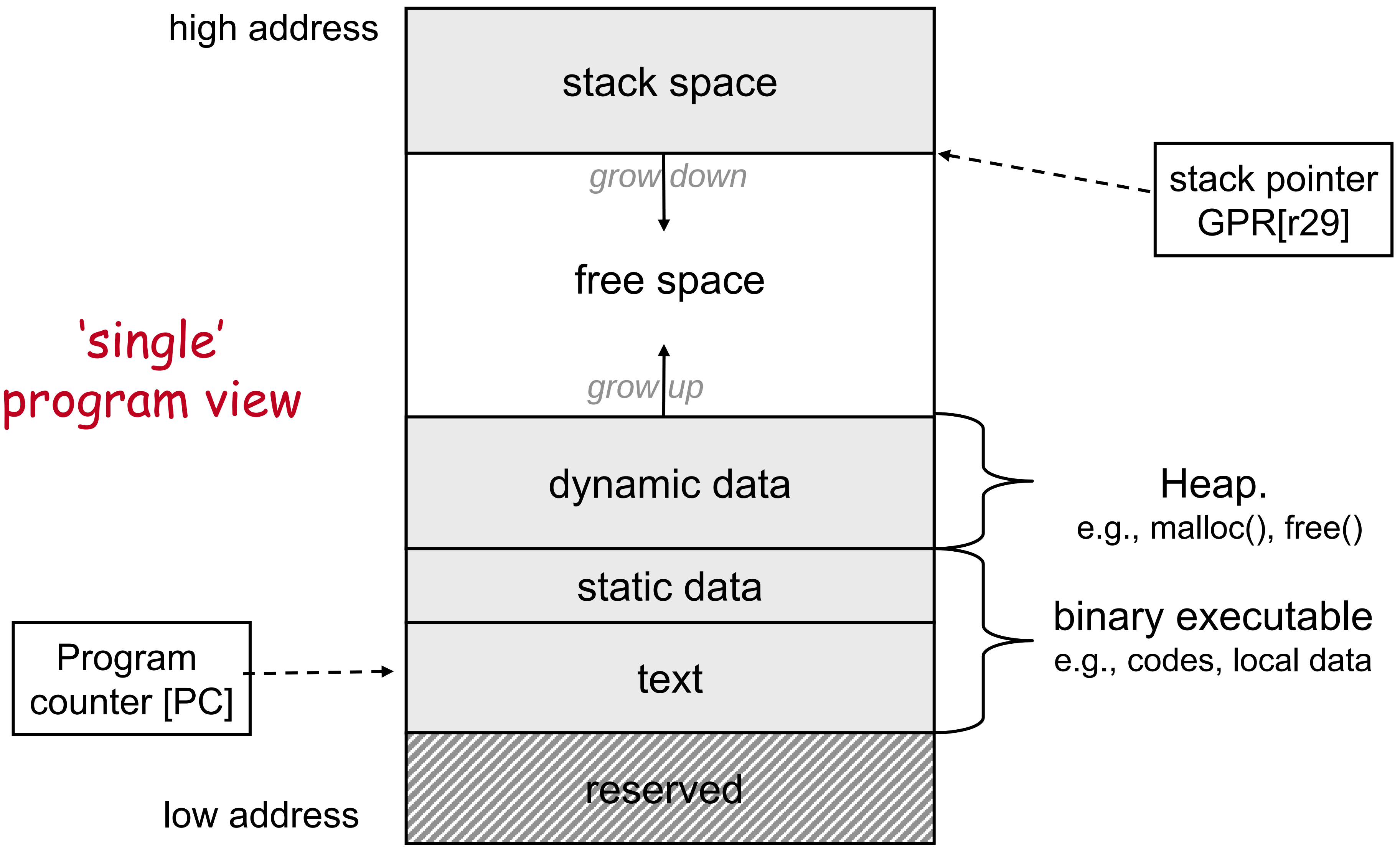
- Caller-Saved Registers
  - Caller says to callee, "If there is anything I care about in these registers, I already saved it myself."
  - Callee says to caller, "Don't count on them staying the same values after I am done.

  → "hey callee! I saved it in my space. use it as you want."

R2000 Register Usage Convention

r0:	always 0	= $zero
r1:	reserved for the assembler	= $at
r2, r3:	function return values	= $v0 ~ $v1
r4~r7:	function call arguments	= $a0 ~ $a3
r8~r15:	"caller-saved" temporaries	= $t0 ~ $t7
r16~r23	"callee-saved" temporaries	= $s0 ~ $s7
r24~r25	"caller-saved" temporaries	= $t8 ~ $t9
r26, r27:	reserved for the operating system	= $k0 ~ $k1
r28:	global pointer	= $gp
r29:	stack pointer	= $sp
r30:	callee-saved temporaries	= $fp
r31:	return address	= $ra

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 <- why?)

$\Big\}$ prologue

.... body of callee (can "nest" additional calls) ....

6. callee loads results to r2, r3
7. callee restores callee-saved registers
8. JR r31

$\Big\}$ epilogue

9. caller restores caller-saved registers
10. 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

Mostly, 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

Question?

Announcements:	*HW #1 will be posted on the course web site soon.*
Reading:	*start reading P&H Ch.3 (stuff for the next class).*
Handouts:	*None*