18-447
Computer Architecture
Lecture 4: ISA Tradeoffs (Continued) and MIPS ISA

Prof. Onur Mutlu
Kevin Chang
Carnegie Mellon University
Spring 2015, 1/21/2015

Agenda for Today

- Finish off ISA tradeoffs
- A quick tutorial on MIPS ISA

- Upcoming schedule:
  - **Lab 1.5 & 2** are out today
  - Friday (1/23): **Lab 1** due
  - Friday (1/23): Recitation
  - Wednesday (1/28): **HW 1** due

Upcoming Readings

- Next week (Microarchitecture):
  - P&H, Chapter 4, Sections 4.1-4.4
  - P&P, revised Appendix C – LC3b datapath and microprogrammed operation

Last Lecture Recap

- Instruction processing style
  - 0, 1, 2, 3 address machines
- Elements of an ISA
  - Instructions, data types, memory organizations, registers, etc
- Addressing modes
- Complex (CISC) vs. simple (RISC) instructions
- Semantic gap
- ISA translation

ISA-level Tradeoffs: Instruction Length

- Fixed length: Length of all instructions the same
  - + Easier to decode single instruction in hardware
  - + Easier to decode multiple instructions concurrently
  - -- Wasted bits in instructions (Why is this bad?)
  - -- Harder-to-extend ISA (how to add new instructions?)
- Variable length: Length of instructions different (determined by opcode and sub-opcode)
  - + Compact encoding (Why is this good?)
    - Intel 432: Huffman encoding (sort of). 6 to 321 bit instructions. How?
  - -- More logic to decode a single instruction
  - -- Harder to decode multiple instructions concurrently
- Tradeoffs
  - Code size (memory space, bandwidth, latency) vs. hardware complexity
  - ISA extensibility and expressiveness vs. hardware complexity
  - Performance? Energy? Smaller code vs. ease of decode

ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location
  - Ditto operand specifiers, immediate values, …
  - Many "RISC" ISAs: Alpha, MIPS, SPARC

  + Easier decode, simpler hardware

  + Enables parallelism: generate target address before knowing the instruction is a branch

  -- Restricts instruction format (fewer instructions?) or wastes space

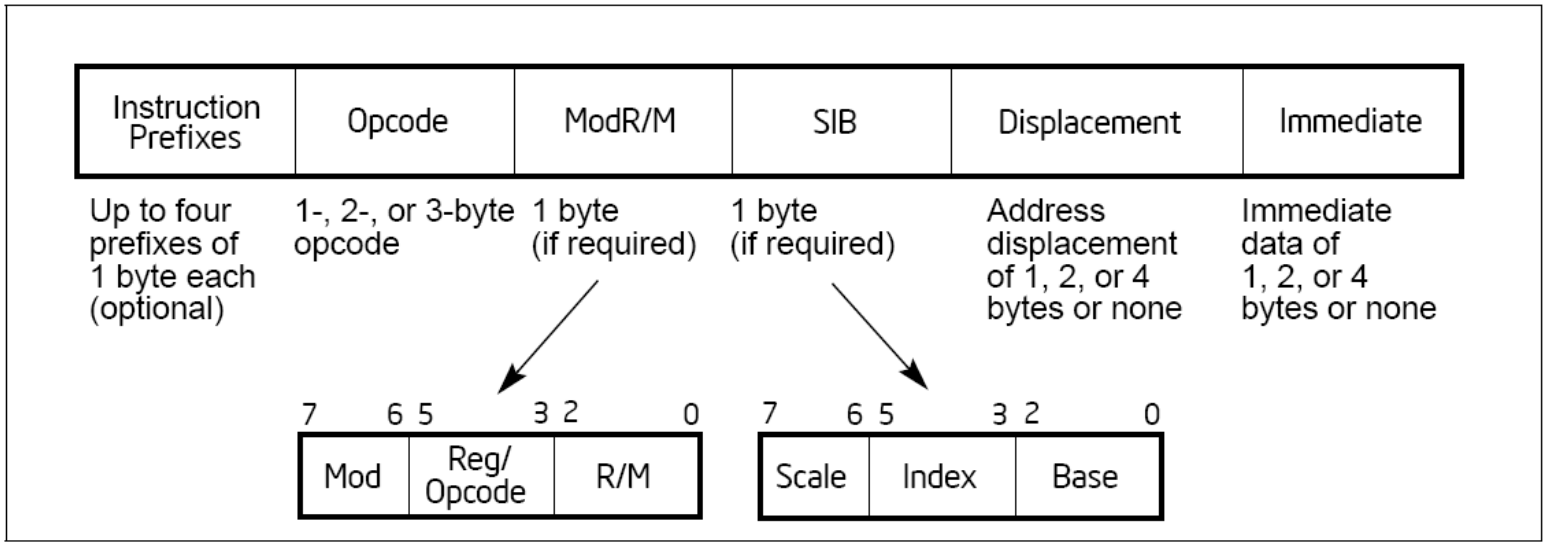
- Non-uniform decode
  - E.g., opcode can be the 1st-7th byte in x86

  + More compact and powerful instruction format

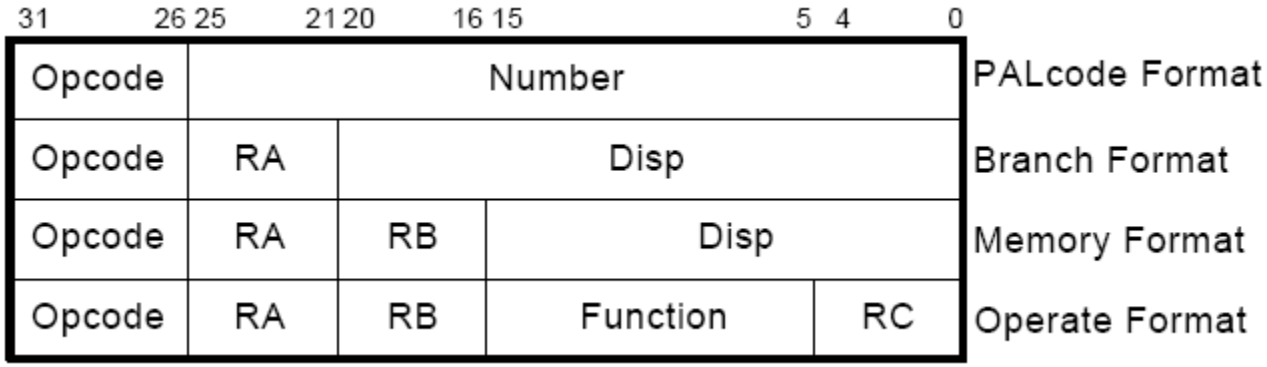
  -- More complex decode logic

x86 vs. Alpha Instruction Formats

- x86:

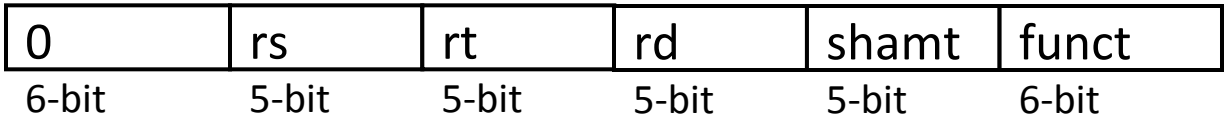


- Alpha:



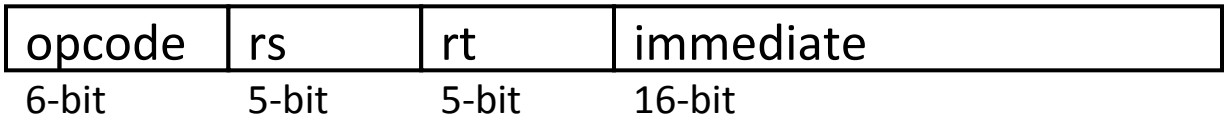
MIPS Instruction Format

- R-type, 3 register operands



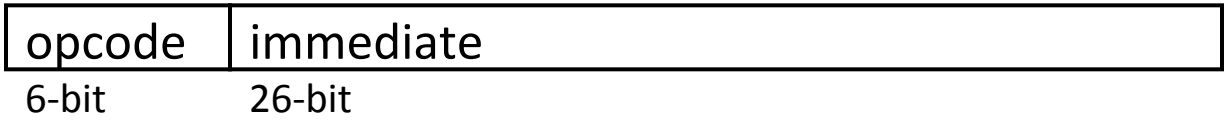
R-type

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



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 easy to extract in hardware

ARM

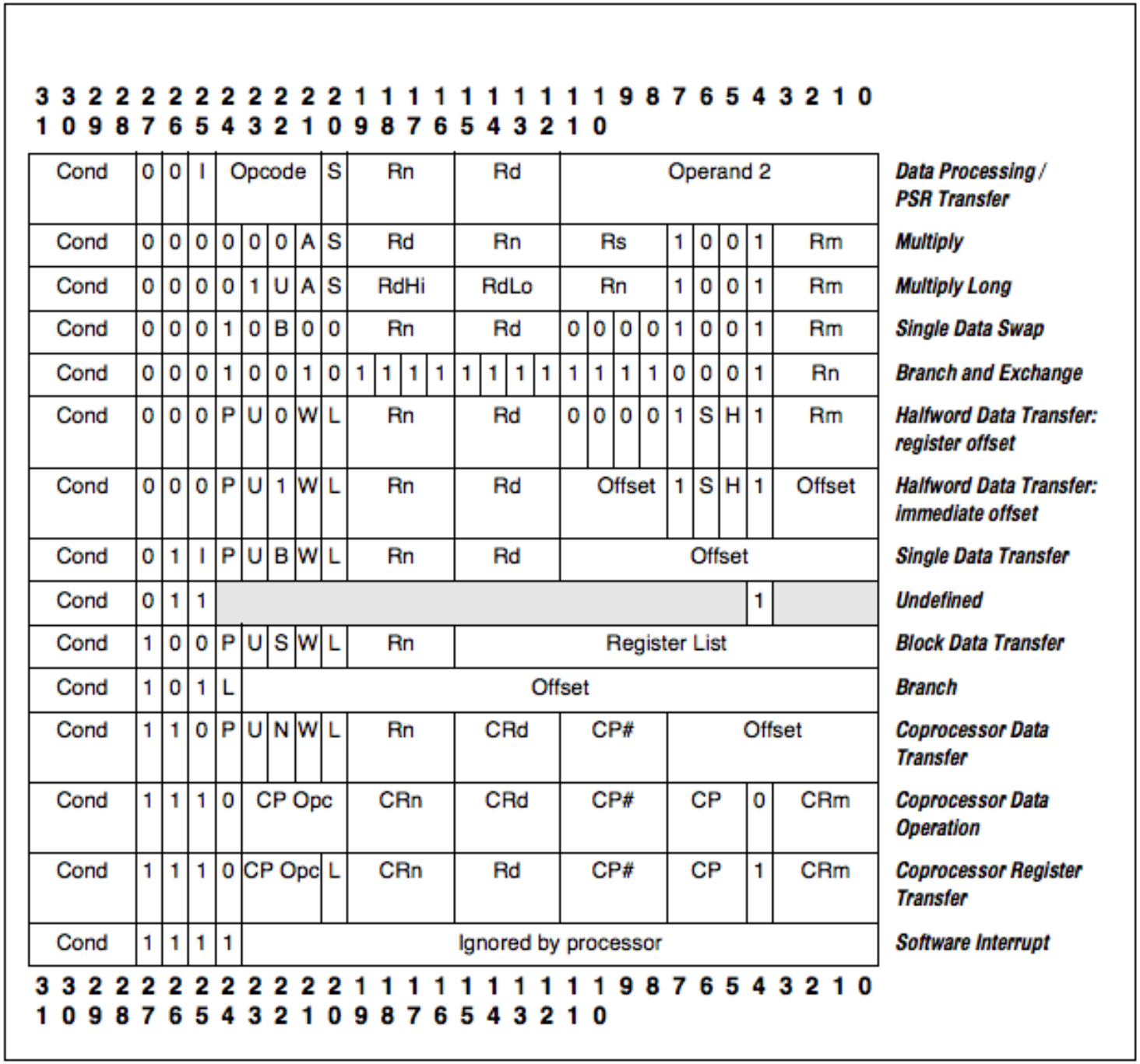


Figure 4-1: ARM instruction set formats

A Note on Length and Uniformity

- Uniform decode usually goes with fixed length

- In a variable length ISA, uniform decode can be a property of instructions of the same length
  - It is hard to think of it as a property of instructions of different lengths

A Note on RISC vs. CISC

- Usually, …

- RISC
  - Simple instructions
  - Fixed length
  - Uniform decode
  - Few addressing modes

- CISC
  - Complex instructions
  - Variable length
  - Non-uniform decode
  - Many addressing modes

ISA-level Tradeoffs: Number of Registers

- Affects:
  - Number of bits used for encoding register address
  - Number of values kept in fast storage (register file)
  - (uarch) Size, access time, power consumption of register file

- Large number of registers:

  + Enables better register allocation (and optimizations) by compiler → fewer saves/restores

  -- Larger instruction size

  -- Larger register file size

ISA-level Tradeoffs: Addressing Modes

- Addressing mode specifies how to obtain an operand of an instruction
  - Register
  - Immediate
  - Memory (displacement, register indirect, indexed, absolute, memory indirect, autoincrement, autodecrement, …)

- More modes:

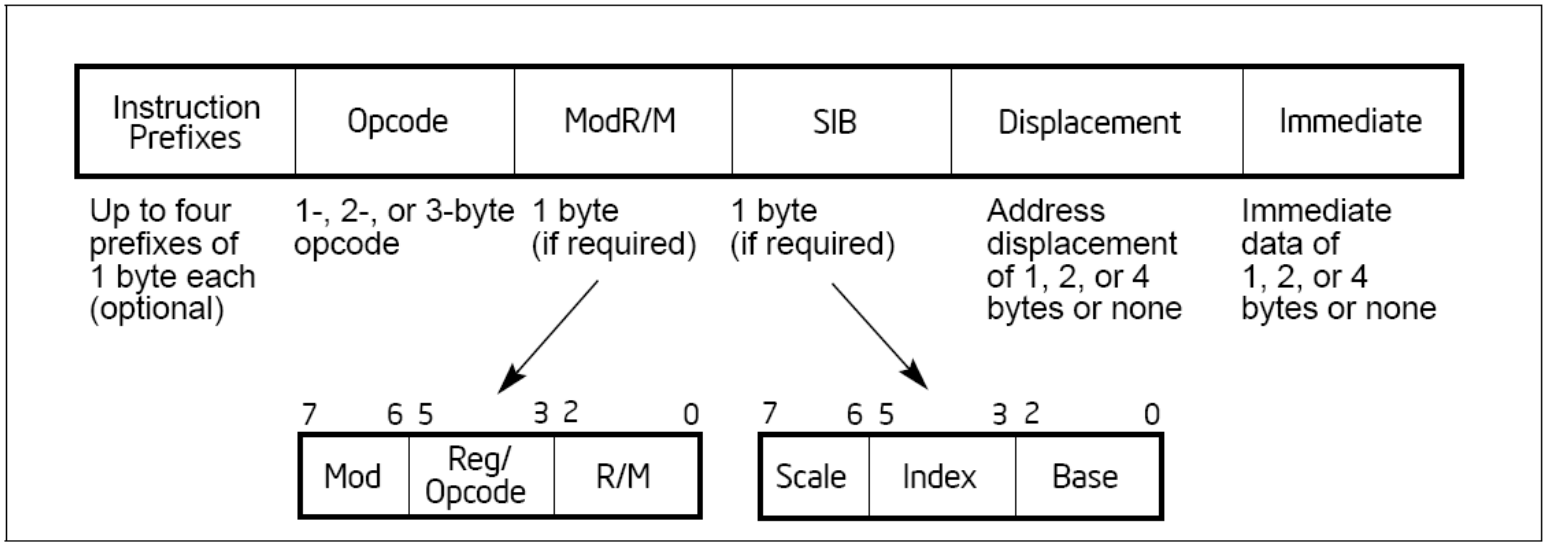
  + help better support programming constructs (arrays, pointer-based accesses)

  -- make it harder for the architect to design

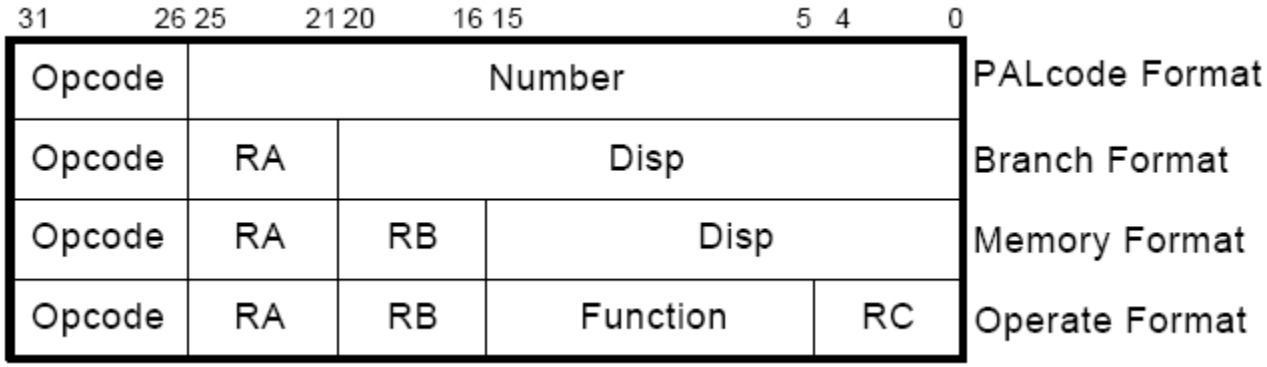
  -- too many choices for the compiler?
    - Many ways to do the same thing complicates compiler design
    - *Wulf, "Compilers and Computer Architecture," IEEE Computer 1981*

x86 vs. Alpha Instruction Formats

- x86:



- Alpha:



register indirect

absolute

SIB + displacement

register + displacement

register

Table 2-2. 32-Bit Addressing Forms with the ModR/M Byte

r8(/r) r16(/r) r32(/r) mm(/r) xmm(/r) (In decimal) /digit (Opcode) (In binary) REG =			AL AX EAX MM0 XMM0 0 000	CL CX ECX MM1 XMM1 1 001	DL DX EDX MM2 XMM2 2 010	BL BX EBX MM3 XMM3 3 011	AH SP ESP MM4 XMM4 4 100	CH BP EBP MM5 XMM5 5 101	DH SI ESI MM6 XMM6 6 110	BH DI EDI MM7 XMM7 7 111
Effective Address	Mod	R/M	Value of ModR/M Byte (in Hexadecimal)							
[EAX]	00	000	00	08	10	18	20	28	30	38
[ECX]		001	01	09	11	19	21	29	31	39
[EDX]		010	02	0A	12	1A	22	2A	32	3A
[EBX]		011	03	0B	13	1B	23	2B	33	3B
[--][--][1]		100	04	0C	14	1C	24	2C	34	3C
disp32[2]		101	05	0D	15	1D	25	2D	35	3D
[ESI]		110	06	0E	16	1E	26	2E	36	3E
[EDI]		111	07	0F	17	1F	27	2F	37	3F
[EAX]+disp8[3]	01	000	40	48	50	58	60	68	70	78
[ECX]+disp8		001	41	49	51	59	61	69	71	79
[EDX]+disp8		010	42	4A	52	5A	62	6A	72	7A
[EBX]+disp8		011	43	4B	53	5B	63	6B	73	7B
[--][--]+disp8		100	44	4C	54	5C	64	6C	74	7C
[EBP]+disp8		101	45	4D	55	5D	65	6D	75	7D
[ESI]+disp8		110	46	4E	56	5E	66	6E	76	7E
[EDI]+disp8		111	47	4F	57	5F	67	6F	77	7F
[EAX]+disp32	10	000	80	88	90	98	A0	A8	B0	B8
[ECX]+disp32		001	81	89	91	99	A1	A9	B1	B9
[EDX]+disp32		010	82	8A	92	9A	A2	AA	B2	BA
[EBX]+disp32		011	83	8B	93	9B	A3	AB	B3	BB
[--][--]+disp32		100	84	8C	94	9C	A4	AC	B4	BC
[EBP]+disp32		101	85	8D	95	9D	A5	AD	B5	BD
[ESI]+disp32		110	86	8E	96	9E	A6	AE	B6	BE
[EDI]+disp32		111	87	8F	97	9F	A7	AF	B7	BF
EAX/AX/AL/MM0/XMM0	11	000	C0	C8	D0	D8	E0	E8	F0	F8
ECX/CX/CL/MM/XMM1		001	C1	C9	D1	D9	E1	E9	F1	F9
EDX/DX/DL/MM2/XMM2		010	C2	CA	D2	DA	E2	EA	F2	FA
EBX/BX/BL/MM3/XMM3		011	C3	CB	D3	DB	E3	EB	F3	FB
ESP/SP/AH/MM4/XMM4		100	C4	CC	D4	DC	E4	EC	F4	FC
EBP/BP/CH/MM5/XMM5		101	C5	CD	D5	DD	E5	ED	F5	FD
ESI/SI/DH/MM6/XMM6		110	C6	CE	D6	DE	E6	EE	F6	FE
EDI/DI/BH/MM7/XMM7		111	C7	CF	D7	DF	E7	EF	F7	FF

Memory

Register

NOTES:

1. The [--][--] nomenclature means a SIB follows the ModR/M byte.
2. The disp32 nomenclature denotes a 32-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is added to the index.
3. The disp8 nomenclature denotes an 8-bit displacement that follows the ModR/M byte (or the SIB byte if one is present) and that is sign-extended and added to the index.

Table 2-3 is organized to give 256 possible values of the SIB byte (in hexadecimal). General purpose registers used as a base are indicated across the top of the table.

indexed (base + index)

scaled (base + index*4)

Table 2-3. 32-Bit Addressing Forms with the SIB Byte

r32 (In decimal) Base – (In binary) Base –			EAX 0 000	ECX 1 001	EDX 2 010	EBX 3 011	ESP 4 100	[*] 5 101	ESI 6 110	EDI 7 111
Scaled Index	SS	Index	Value of SIB Byte (in Hexadecimal)							
[EAX]	00	000	00	01	02	03	04	05	06	07
[ECX]		001	08	09	0A	0B	0C	0D	0E	0F
[EDX]		010	10	11	12	13	14	15	16	17
[EBX]		011	18	19	1A	1B	1C	1D	1E	1F
none		100	20	21	22	23	24	25	26	27
[EBP]		101	28	29	2A	2B	2C	2D	2E	2F
[ESI]		110	30	31	32	33	34	35	36	37
[EDI]		111	38	39	3A	3B	3C	3D	3E	3F
[EAX*2]	01	000	40	41	42	43	44	45	46	47
[ECX*2]		001	48	49	4A	4B	4C	4D	4E	4F
[EDX*2]		010	50	51	52	53	54	55	56	57
[EBX*2]		011	58	59	5A	5B	5C	5D	5E	5F
none		100	60	61	62	63	64	65	66	67
[EBP*2]		101	68	69	6A	6B	6C	6D	6E	6F
[ESI*2]		110	70	71	72	73	74	75	76	77
[EDI*2]		111	78	79	7A	7B	7C	7D	7E	7F
[EAX*4]	10	000	80	81	82	83	84	85	86	87
[ECX*4]		001	88	89	8A	8B	8C	8D	8E	8F
[EDX*4]		010	90	91	92	93	94	95	96	97
[EBX*4]		011	98	89	9A	9B	9C	9D	9E	9F
none		100	A0	A1	A2	A3	A4	A5	A6	A7
[EBP*4]		101	A8	A9	AA	AB	AC	AD	AE	AF
[ESI*4]		110	B0	B1	B2	B3	B4	B5	B6	B7
[EDI*4]		111	B8	B9	BA	BB	BC	BD	BE	BF
[EAX*8]	11	000	C0	C1	C2	C3	C4	C5	C6	C7
[ECX*8]		001	C8	C9	CA	CB	CC	CD	CE	CF
[EDX*8]		010	D0	D1	D2	D3	D4	D5	D6	D7
[EBX*8]		011	D8	D9	DA	DB	DC	DD	DE	DF
none		100	E0	E1	E2	E3	E4	E5	E6	E7
[EBP*8]		101	E8	E9	EA	EB	EC	ED	EE	EF
[ESI*8]		110	F0	F1	F2	F3	F4	F5	F6	F7
[EDI*8]		111	F8	F9	FA	FB	FC	FD	FE	FF

NOTES:

1. The [*] nomenclature means a disp32 with no base if the MOD is 00B. Otherwise, [*] means disp8 or disp32 + [EBP]. This provides the following address modes:

MOD bits	Effective Address
00	[scaled index] + disp32
01	[scaled index] + disp8 + [EBP]
10	[scaled index] + disp32 + [EBP]

X86 SIB-D Addressing Mode

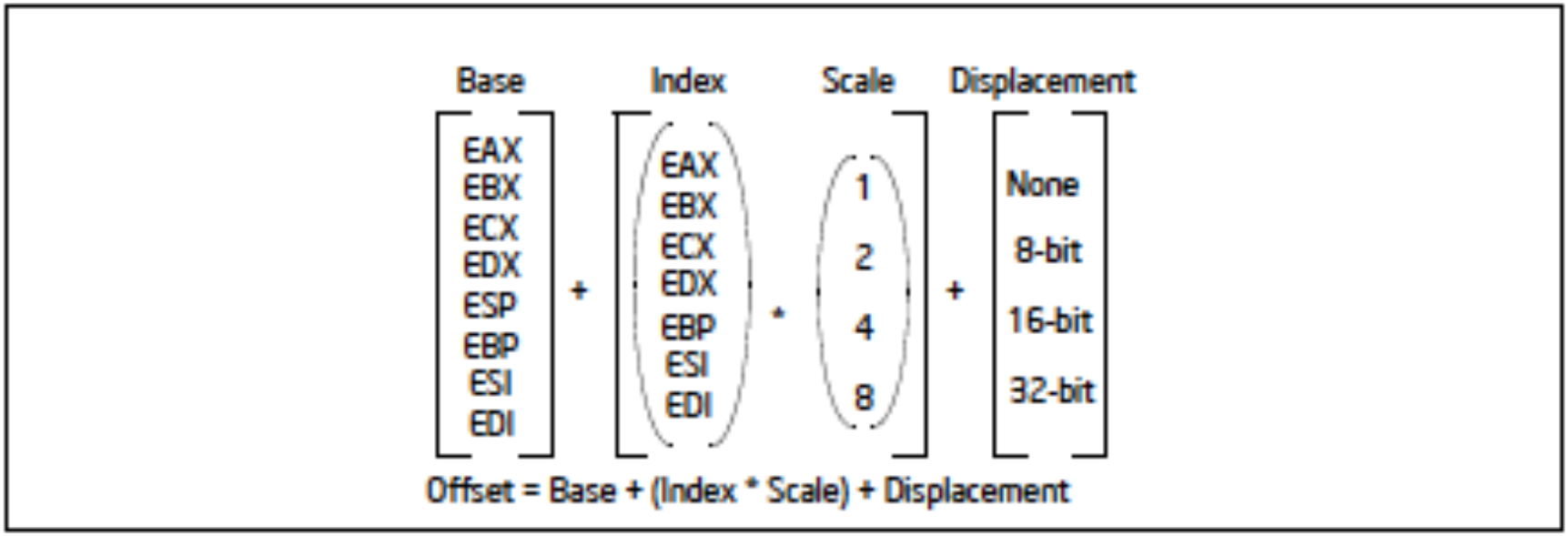


Figure 3-11. Offset (or Effective Address) Computation

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

The following addressing modes suggest uses for common combinations of address components.

- **Displacement** — A displacement alone represents a direct (uncomputed) offset to the operand. Because the displacement is encoded in the instruction, this form of an address is sometimes called an absolute or static address. It is commonly used to access a statically allocated scalar operand.

  Static address

- **Base** — A base alone represents an indirect offset to the operand. Since the value in the base register can change, it can be used for dynamic storage of variables and data structures.

  Dynamic storage

- **Base + Displacement** — A base register and a displacement can be used together for two distinct purposes:
  - As an index into an array when the element size is not 2, 4, or 8 bytes—The displacement component encodes the static offset to the beginning of the array. The base register holds the results of a calculation to determine the offset to a specific element within the array.

    Arrays
  - To access a field of a record: the base register holds the address of the beginning of the record, while the displacement is a static offset to the field.

    Records

  An important special case of this combination is access to parameters in a procedure activation record. A procedure activation record is the stack frame created when a procedure is entered. Here, the EBP register is the best choice for the base register, because it automatically selects the stack segment. This is a compact encoding for this common function.

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

X86 Manual: Suggested Uses of Addressing Modes

- **(Index * Scale) + Displacement** — This address mode offers an efficient way to index into a static array when the element size is 2, 4, or 8 bytes. The displacement locates the [illegible] holds the subscript of the desired array element, and the processor automatically converts the subscript into an index by applying the scaling factor.

Static arrays w/ fixed-size elements

- **Base + Index + Displacement** — Using two registers together supports either a two-dimensional ar[illegible]acement holds the address of the beginning of the array) or one of several instances of an array of records (the displacement is an offset to a field within the record).

2D arrays

- **Base + (Index * Scale) + Displacement** — Using all the addressing components together [illegible]ent indexing of a two-dimensional array when the elements of the a[illegible], or 8 bytes in size.

2D arrays

x86 Manual Vol. 1, page 3-22 -- see course resources on website
Also, see Section 3.7.3 and 3.7.5

Other Example ISA-level Tradeoffs

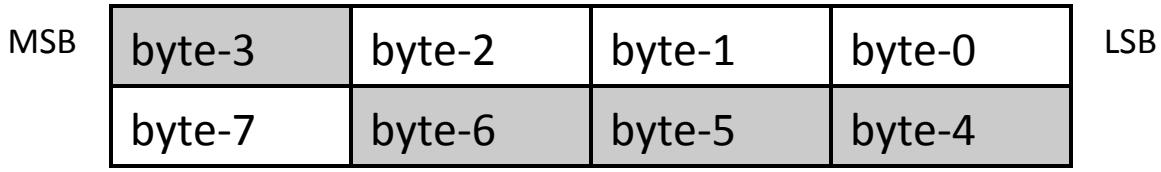
- Condition codes vs. not
- VLIW vs. single instruction
- Precise vs. imprecise exceptions
- Virtual memory vs. not
- Unaligned access vs. not
- Hardware interlocks vs. software-guaranteed interlocking
- Software vs. hardware managed page fault handling
- Cache coherence (hardware vs. software)
- …

Back to Programmer vs. (Micro)architect

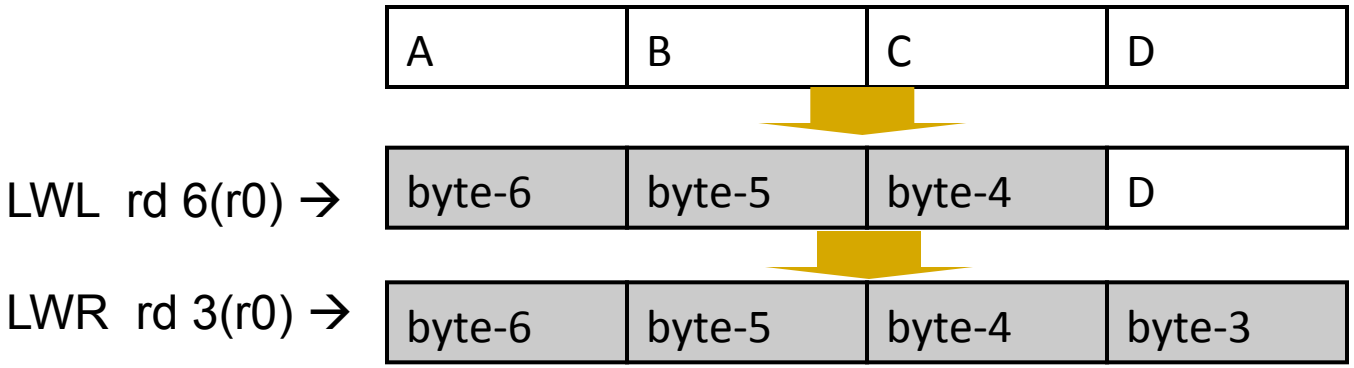
- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compiler/programmer needs to align data

MIPS: Aligned Access



- LW/SW alignment restriction: 4-byte word-alignment
  - not designed to fetch memory bytes not within a word boundary
  - not designed to rotate unaligned bytes into registers
- Provide separate opcodes for the "infrequent" case



  - LWL/LWR is slower
  - Note LWL and LWR still fetch within word boundary

X86: Unaligned Access

- LD/ST instructions automatically align data that spans a "word" boundary
- Programmer/compiler does not need to worry about where data is stored (whether or not in a word-aligned location)

4.1.1 Alignment of Words, Doublewords, Quadwords, and Double Quadwords

Words, doublewords, and quadwords do not need to be aligned in memory on natural boundaries. The natural boundaries for words, double words, and quadwords are even-numbered addresses, addresses evenly divisible by four, and addresses evenly divisible by eight, respectively. However, to improve the performance of programs, data structures (especially stacks) should be aligned on natural boundaries whenever possible. The reason for this is that the processor requires two memory accesses to make an unaligned memory access; aligned accesses require only one memory access. A word or doubleword operand that crosses a 4-byte boundary or a quadword operand that crosses an 8-byte boundary is considered unaligned and requires two separate memory bus cycles for access.

X86: Unaligned Access

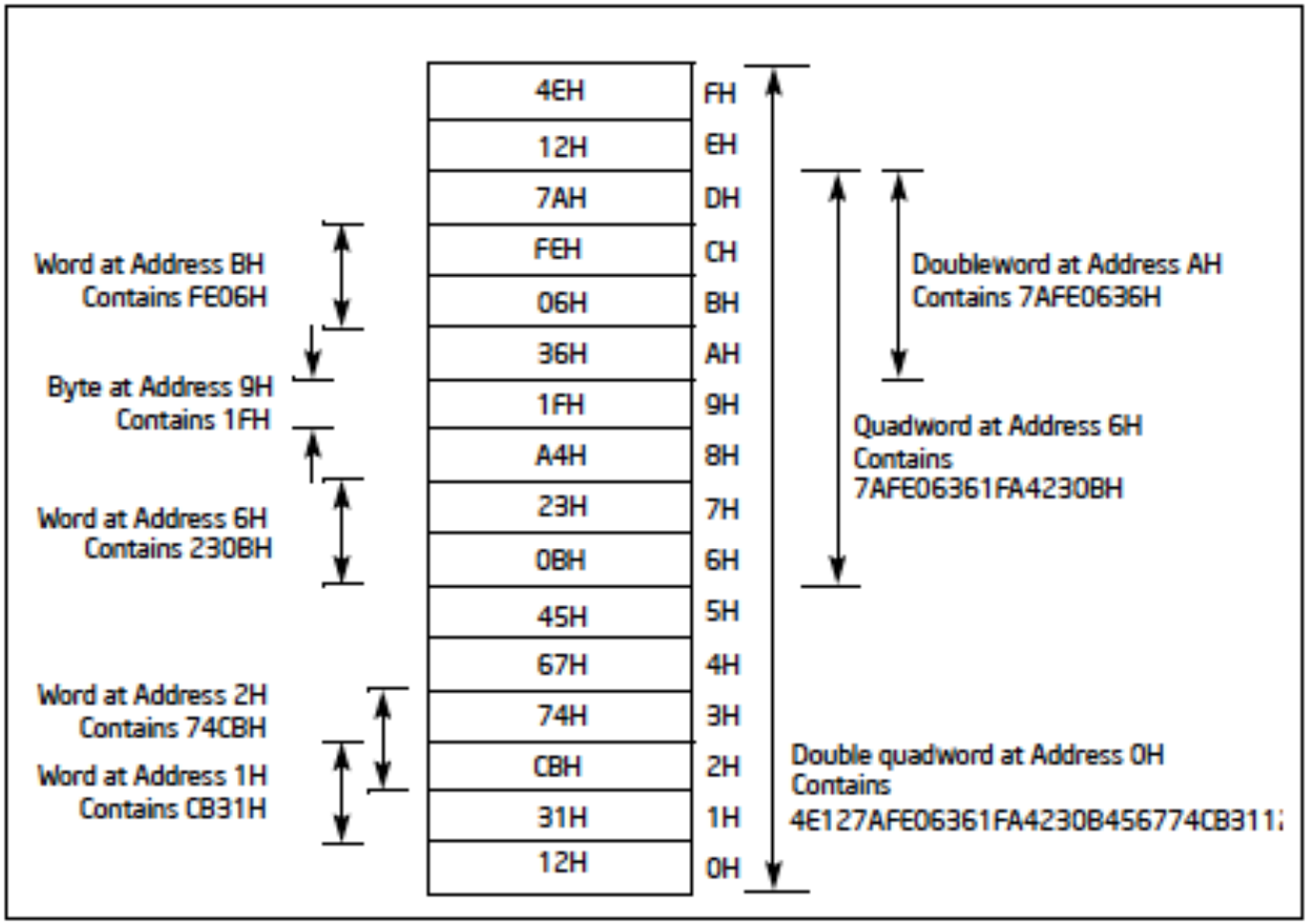


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

What About ARM?

- https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf
  - Section A2.8

Aligned vs. Unaligned Access

- Pros of having no restrictions on alignment
- Cons of having no restrictions on alignment
- Filling in the above: an exercise for you…

18-447 MIPS ISA

James C. Hoe
Dept of ECE, CMU

MIPS R2000 Program Visible State

Program Counter

32-bit memory address of the current instruction

Note r0=0
r1
r2
General Purpose Register File 32 32-bit words named r0...r31

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

Memory

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

32-bit address

(there is some magic going on)

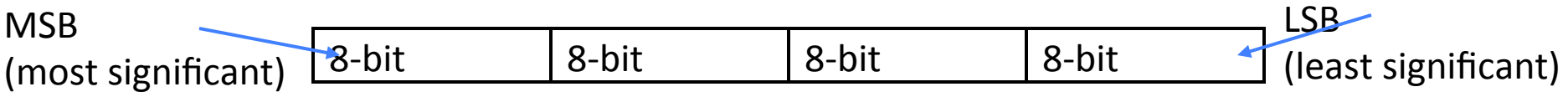
Data Format

- 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 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

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

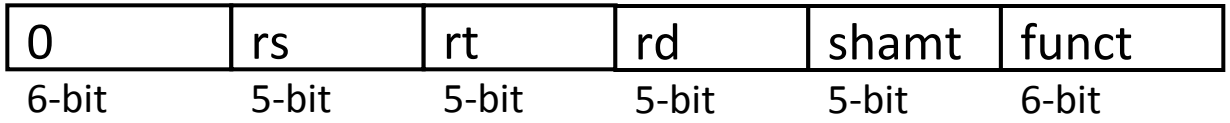
pointer points to the little end

- What difference does it make?

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

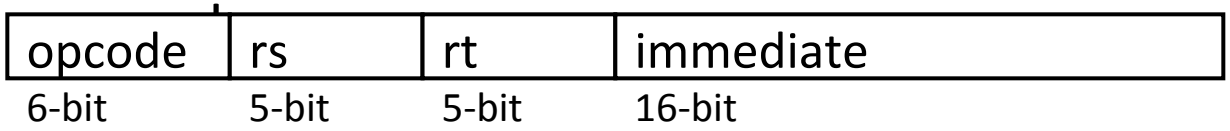
Instruction Formats

- 3 simple formats
  - R-type, 3 register operands



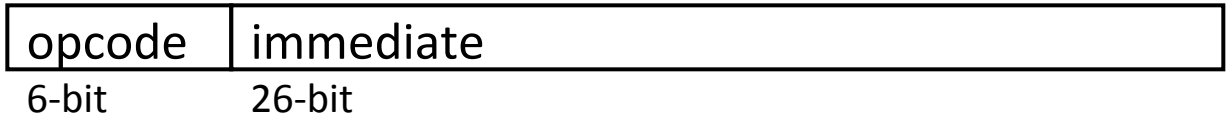
R-type

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



I-type

  - J-type, 26-bit immediate operand



J-type

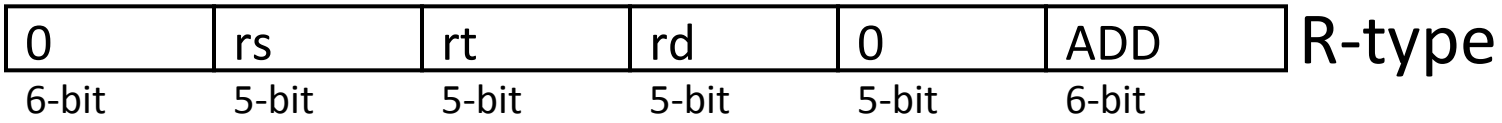
- Simple Decoding
  - 4 bytes per instruction, regardless of format
  - must be 4-byte aligned (2 lsb of PC must be 2b'00)
  - format and fields readily extractable

ALU Instructions

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

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

- Machine encoding



- Semantics
  - GPR[rd] ← GPR[rs] + GPR[rt]
  - PC ← 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

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]

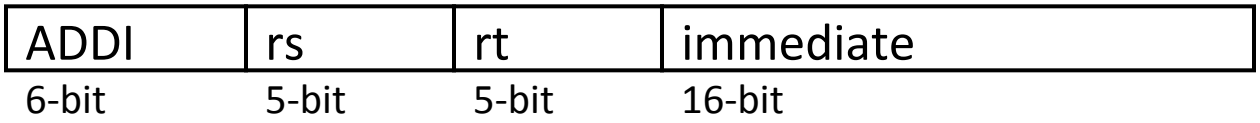
What patterns do you see? Why are they there?

ALU Instructions

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

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

- Machine encoding



I-type

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

Reg-Immed Instruction Encoding

31...29 \ 28...26 (Opcode)	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

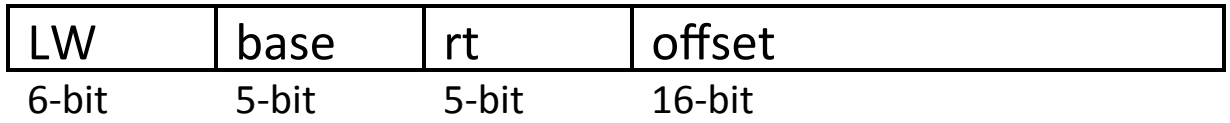
```
add r_temp r_g r_h     # r_temp = g+h
add r_f r_i r_j        # r_f = i+j
sub r_f r_temp r_f     # f = r_temp - r_f
```

Load Instructions

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

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

- Machine encoding



I-type

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

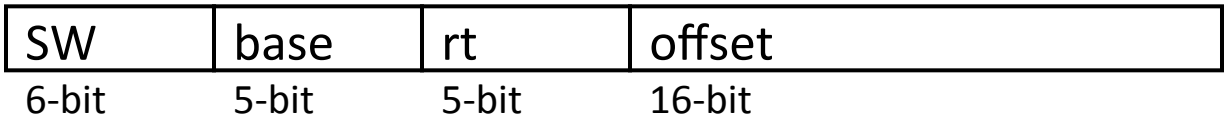
    What if you want to load an unaligned word?
  - MMU exceptions

Store Instructions

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

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

- Machine encoding



I-type

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

Assembly Programming 201

- E.g. High-level Code

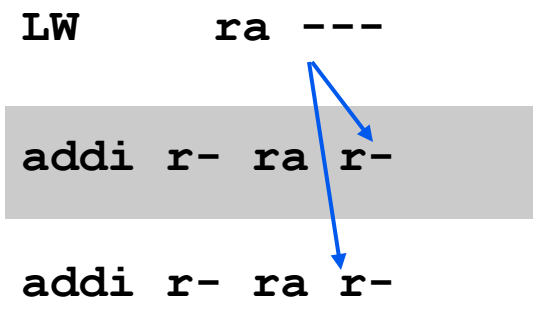
```
A[ 8 ] = h + A[ 0 ]
```

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]
```

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 redefined LW to complete atomically)

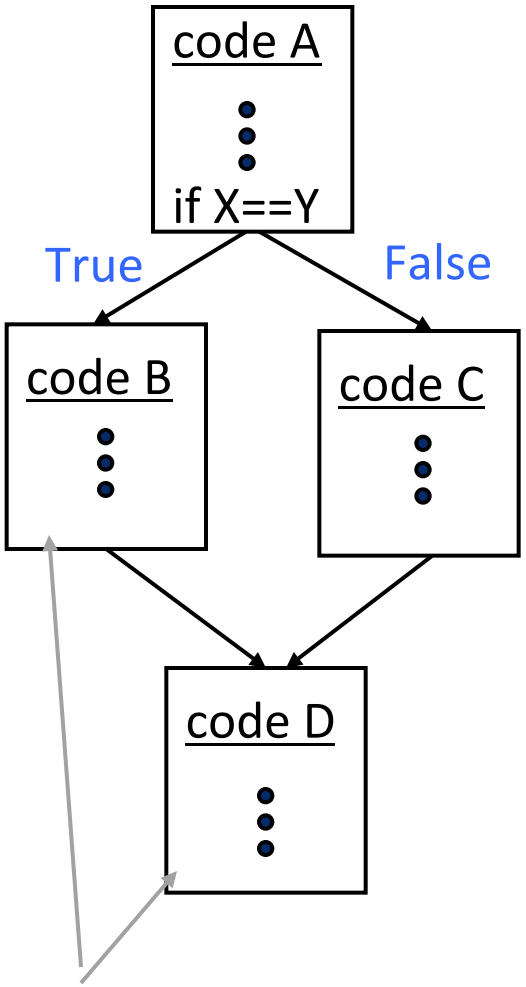
*BTW, notice that latency is defined in "instructions" not cyc. or sec.

Control Flow Instructions

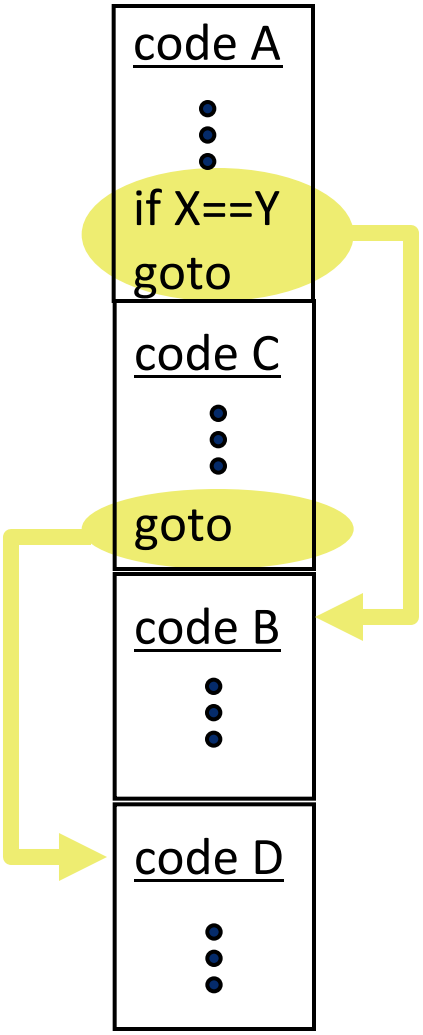
- C-Code

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

Control Flow Graph



Assembly Code (linearized)



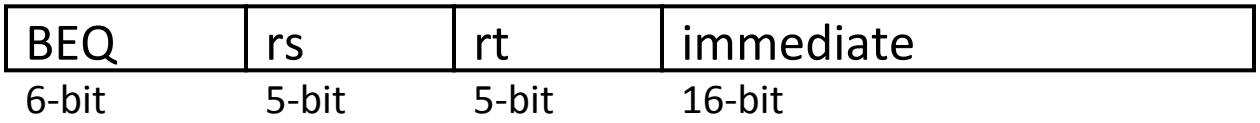
these things are called basic blocks

(Conditional) Branch Instructions

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

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

- Machine encoding



I-type

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

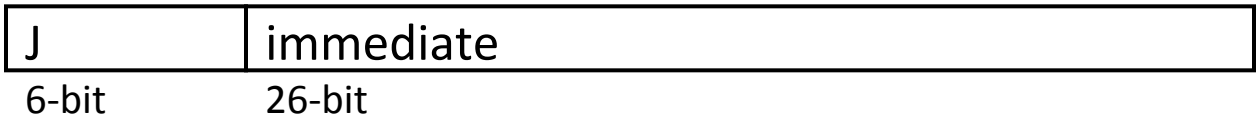
PC + 4 w/ branch delay slot

Why isn't there a BLE or BGT instruction?

Jump Instructions

- Assembly

  J immediate$_{26}$
- Machine encoding



J-type

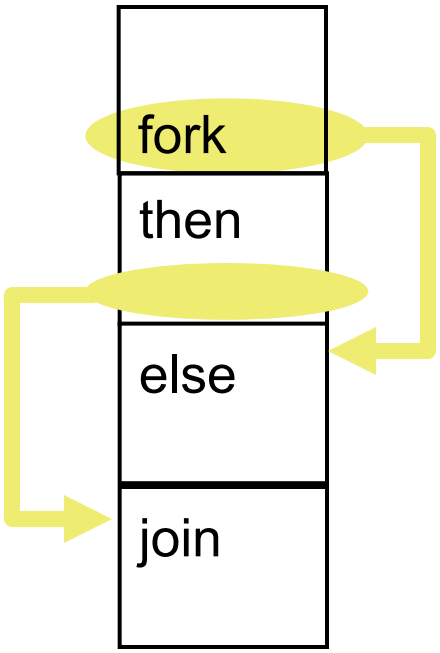
- Semantics
  - target = PC[31:28]x2^{28} $|_{\text{bitwise-or}}$ zero-extend(immediate)x4
  - PC ← target
- How far can you jump?
- Variations
  - Jump and Link
  - Jump Registers

PC + 4 w/ branch delay slot

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
        . . . .
```

Branch Delay Slots

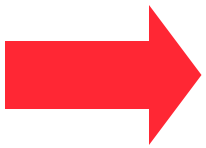
- R2000 branch instructions also have an architectural latency of 1 instructions
  - the instruction immediately after a branch is always executed (in fact PC-offset is computed from the delay slot instruction)
  - branch target takes effect on the 2nd instruction

```
      bne r_i r_j L1

      add r_e r_g r0
      j L2

L1:   add r_e r_h r0

L2:   add r_f r_e r0
      . . . .
```



```
      bne r_i r_j L1
      nop

      j L2
      add r_e r_g r0
L1:   add r_e r_h r0

L2:   add r_f r_e 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 = PC[31:28]x2^{28} $|_{\text{bitwise-or}}$ zero-extend(immediate)x4
  - PC ← 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 ← 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** $\rightarrow_{\text{call}}$ **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

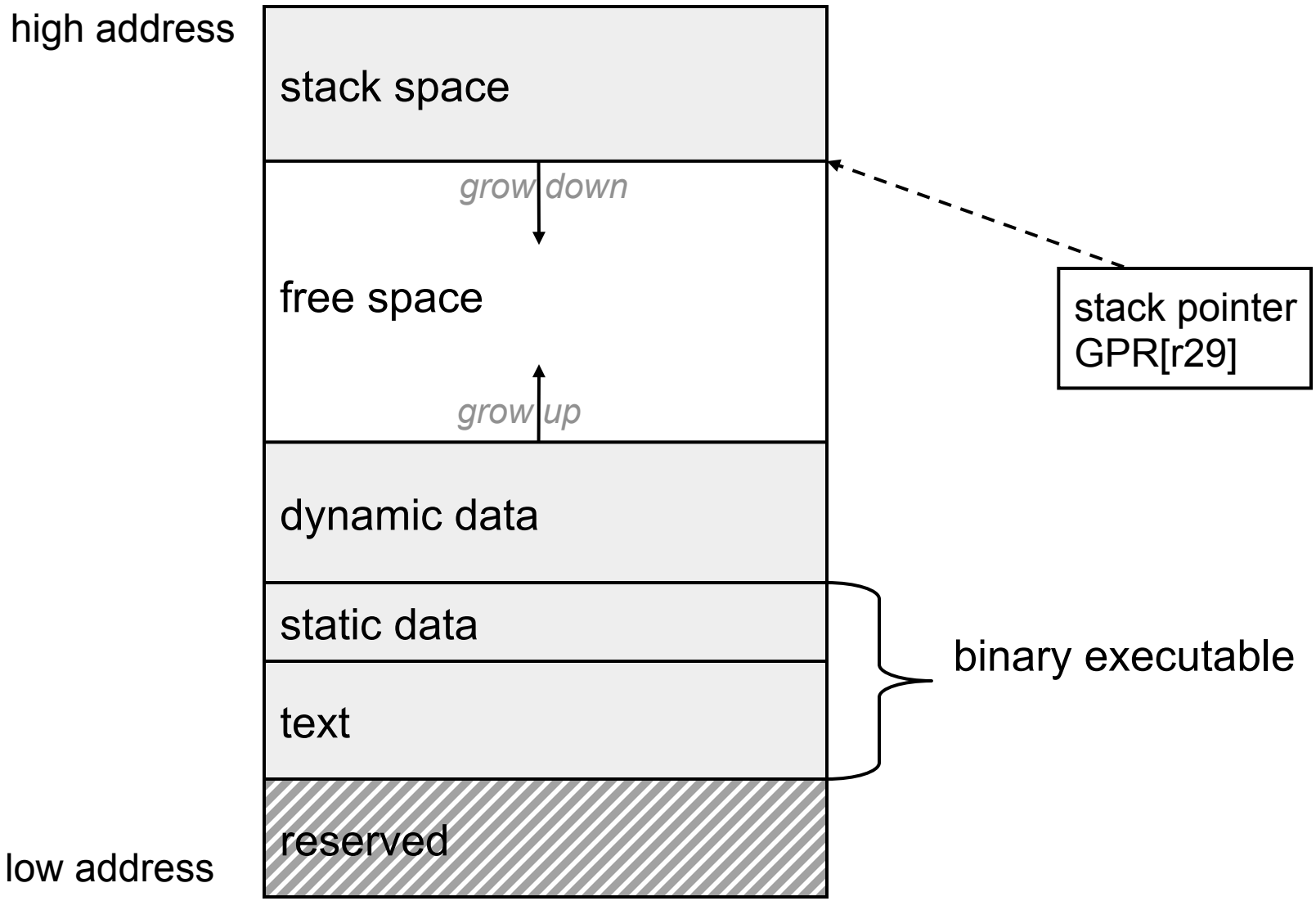
- Callee-Saved Registers
  - Caller says to callee, "The values of these registers should not change when you return to me."
  - Callee says, "If I need to use these registers, I promise to save the old values to memory first and restore them before I return to you."

- 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)

prologue

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

6. callee loads results to r2, r3
7. callee restores saved register values
8. JR r31

epilogue

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