BGU 361-1-4201
Computer Architecture
Lecture 2: ISA Tradeoffs (Continued) and RISC-V ISA
Lecturer: Dr. Guy Tel-Zur

Based on lectures and slides
by Prof. Onur Mutlu
Kevin Chang
Carnegie Mellon University
Spring 2015

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 (superscalar processors – next slide)
- -- 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 מעבדים סופרסקלריים יותר מורכבים
- - Harder to decode multiple instructions concurrently

Tradeoffs

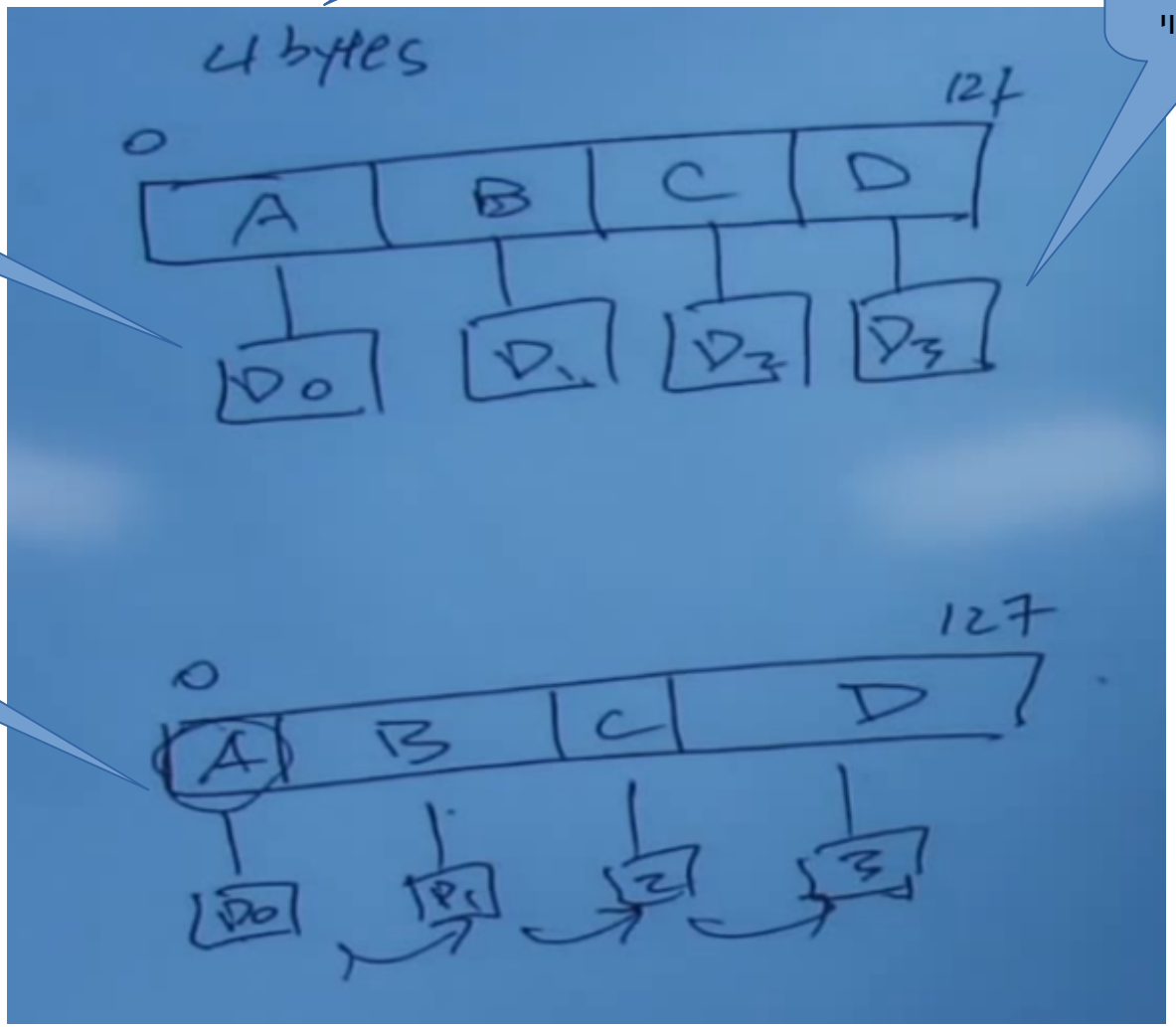
- Code size (memory space, bandwidth, latency) vs. hardware complexity
- ISA extensibility and expressiveness vs. hardware complexity
- Performance? Energy? Smaller code vs. ease of decode
- ביצועים – אם נסתכל על יחס החישוב לתקשורת: אורך משתנה של פקודות מיטיב עם הזכרון ולכן יתכן שיש יתרון לאורך משתנה על פני אורך קבוע.

4 הוראות באורך 4 בייטים

4 מפענחים העובדים באופן בלתי תלוי

עם הוראות **בגודל קבוע** ניתן למקבל את עבודת ה-instruction decoders

עם הוראות ב**גודל משתנה** יש תלות סריאלית בעבודת ה-instruction decoders



סופר-סקאלר

מתוך התיעוד של iAPX432

GDP = General Data Processor

INSTRUCTION OBJECTS

The GDP represents a procedure as one or more instruction objects (see Figure 2-1). When a procedure (an instruction object) is called, the GDP requires certain information for the context object that represents the call. This information is contained in an Instruction Object Header in the first eight bytes of the instruction object's data part. The remainder of the data part can contain instructions.

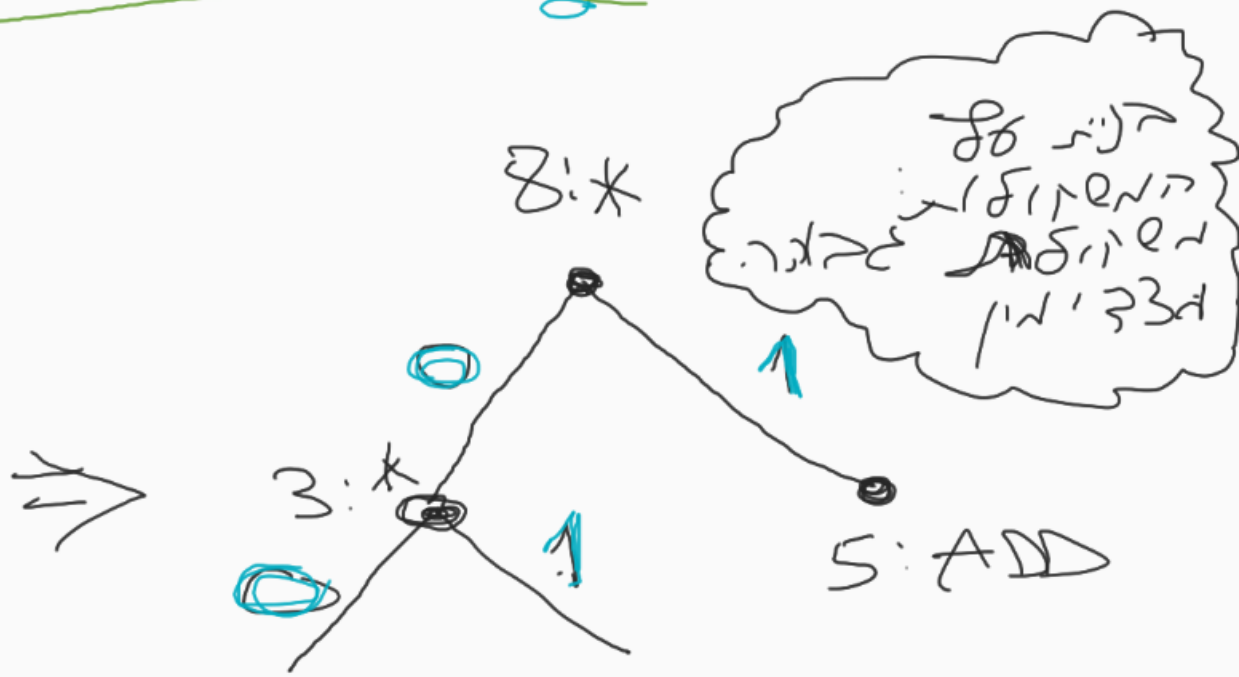
GDP instructions are not aligned on byte or word boundaries and are varying-length sequences of bits. Instruction fields are frequency encoded (Huffman encoded) so that more frequent operation codes, formats, classes, and addressing modes are encoded in fewer bits.

נעשה פרופיילינג של כל הפקודות ונבדוק את תדירות ההופעה שלהן

http://www.bitsavers.org/components/intel/iAPX_432/171860-004_iAPX_432_General_Data_Processor_Architecture_Reference_Manual_Feb84.pdf

Huffman Encoding

פקודה	תדירות השימוש
MULT	1
DIV	2
ADD	5



קידוד:

	הקידוד
ADD	1
DIV	01
MULT	00

על פי הקידוד הבא

ISA-level Tradeoffs: Uniform Decode

- Uniform decode: Same bits in each instruction correspond to the same meaning
  - Opcode is always in the same location – כמו במיפס
  - 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

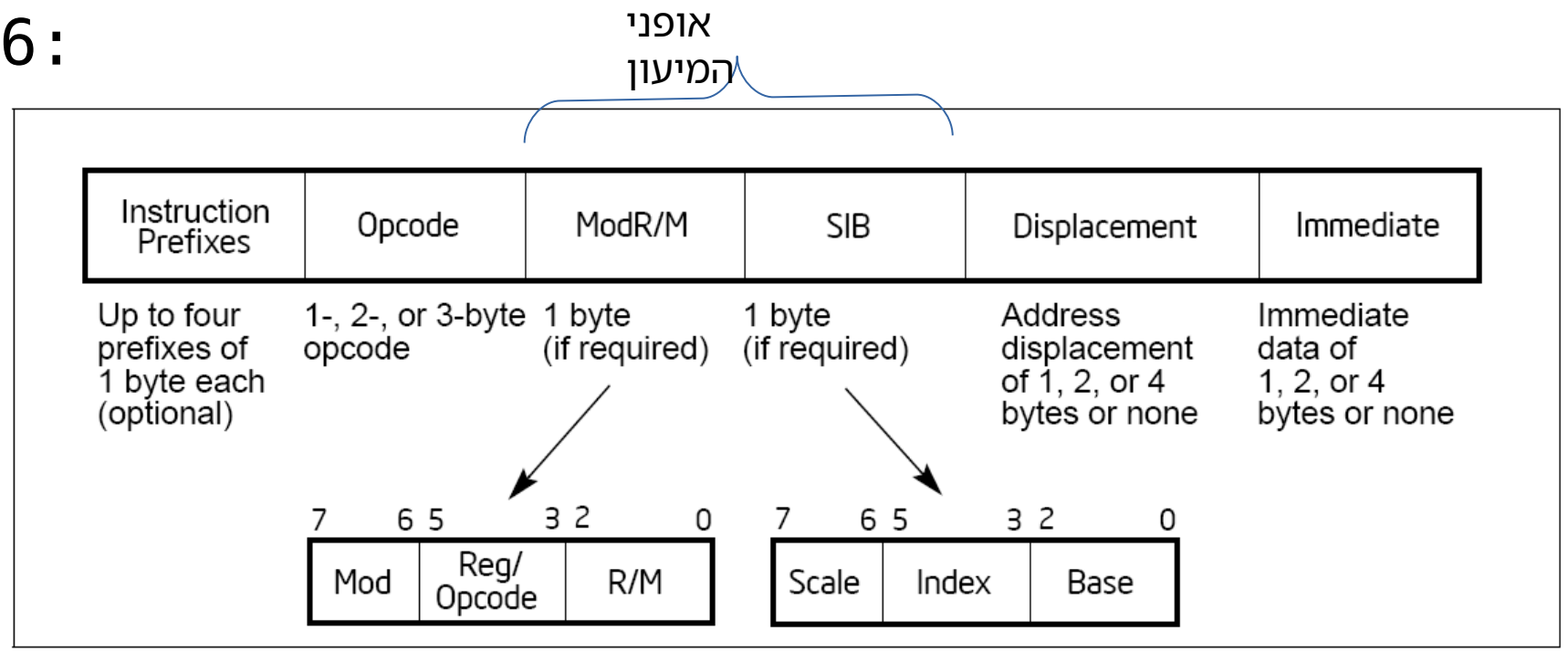
- Non-uniform decode
  - E.g., opcode can be the 1st-7th byte in x86 – קיים טווח של גדלים וזה מאפשר מבנה יותר קומפקטי

  \+ More compact and powerful instruction format

  \- More complex decode logic

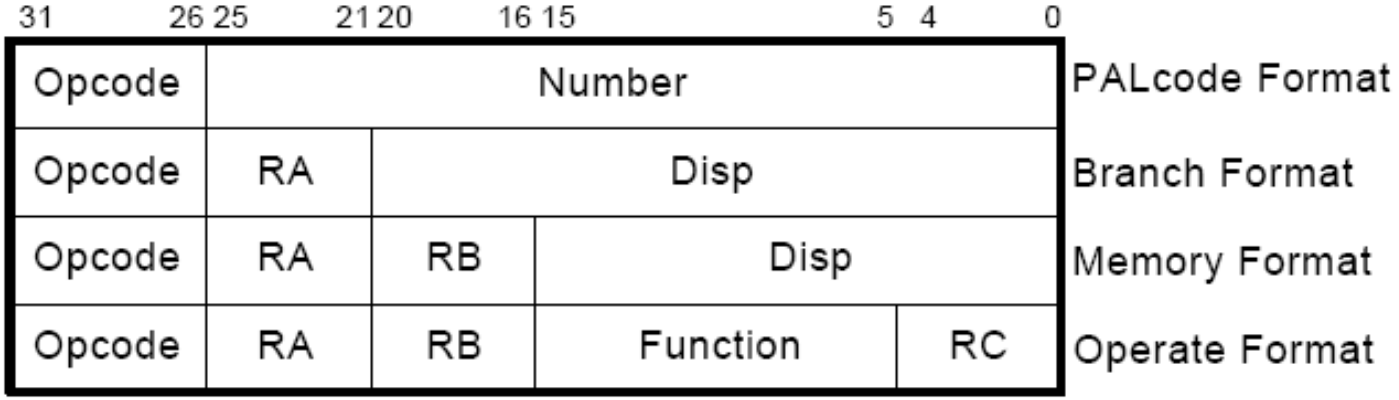
x86 vs. Alpha Instruction Formats

- x86:



- Alpha:

בזכות המבנה הקבוע מתאפשר מיקבול: בזמן שנעשה decode של הפקודה ניתן בצורה ספקולטיבית להעריך את התוכן של 21 הביטים בחלק השני של הפקודה



The SIB byte is structured as follows:
[Scale] [Index] [Base]

* Scale (2 bits): This field specifies a scaling factor that multiplies the value in the index register.
    - The possible values are: 00: Scale = 1, 01: Scale = 2, 10: Scale = 4, 11: Scale = 8
    - The scaled index register is used to access elements of arrays with different element sizes.

* Index (3 bits):
    - This field specifies an index register (e.g., EAX, ECX, EDX, EBX, ESI, EDI).
    - The value in the index register is multiplied by the scale factor.

* Base (3 bits):
    - This field specifies a base register (e.g., EAX, ECX, EDX, EBX, ESI, EDI, ESP, EBP).7
    - The value in the base register is added to the scaled index value and any displacement to calculate the final memory address.

* Address Calculation. The final memory address is calculated as follows:

Address = Base + (Index * Scale) + Displacement

- `Base`: The value in the base register (or 0 if no base register).
- `Index`: The value in the index register (or 0 if no index register).
- `Scale`: The scaling factor (1, 2, 4, or 8).
- `Displacement`: An optional signed displacement value (8, 32 bit).

* Example:

Consider the following assembly instruction:

mov eax, [ebx + ecx * 4]

In this case:

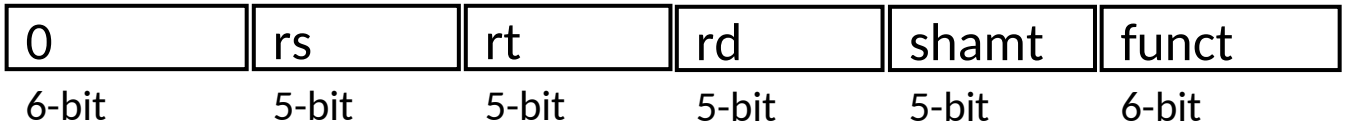
- `ebx` is the base register.
- `ecx` is the index register.
- The scale factor is 4.

The corresponding SIB byte would be:

- Scale: 10 (which is 4)
- Index: The encoding for `ecx`
- Base: The encoding for `ebx`

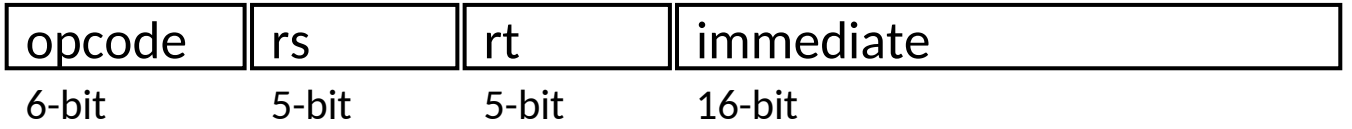
MIPS Instruction Format

- R-type, 3 register operands



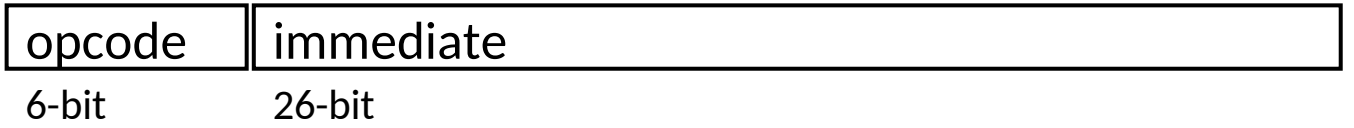
R-type

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



I-type

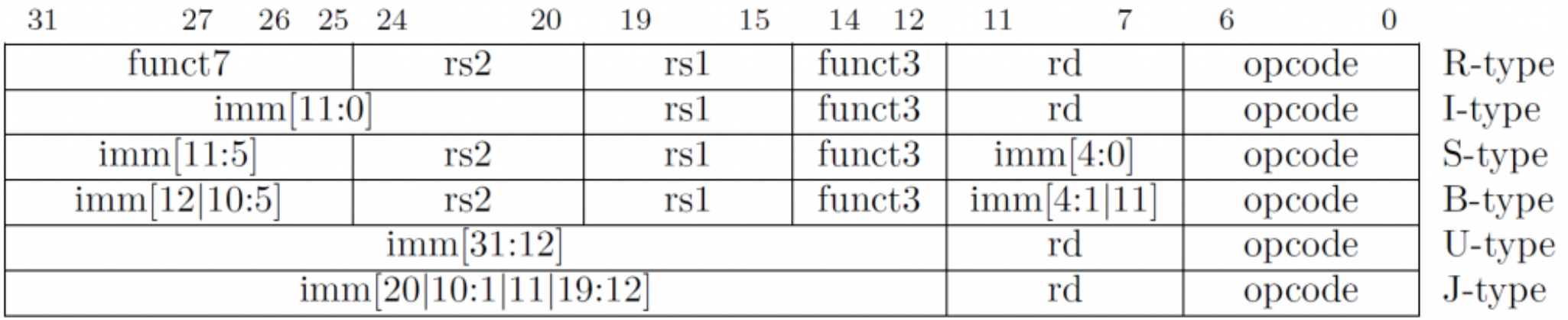
- J-type, 26-bit immediate operand



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

RISC-V Instruction Format



R - Register-register operations

I - Short immediates and loads

S - Stores

B - Conditional branches

U - Long immediates

J - Unconditional jumps

ARM

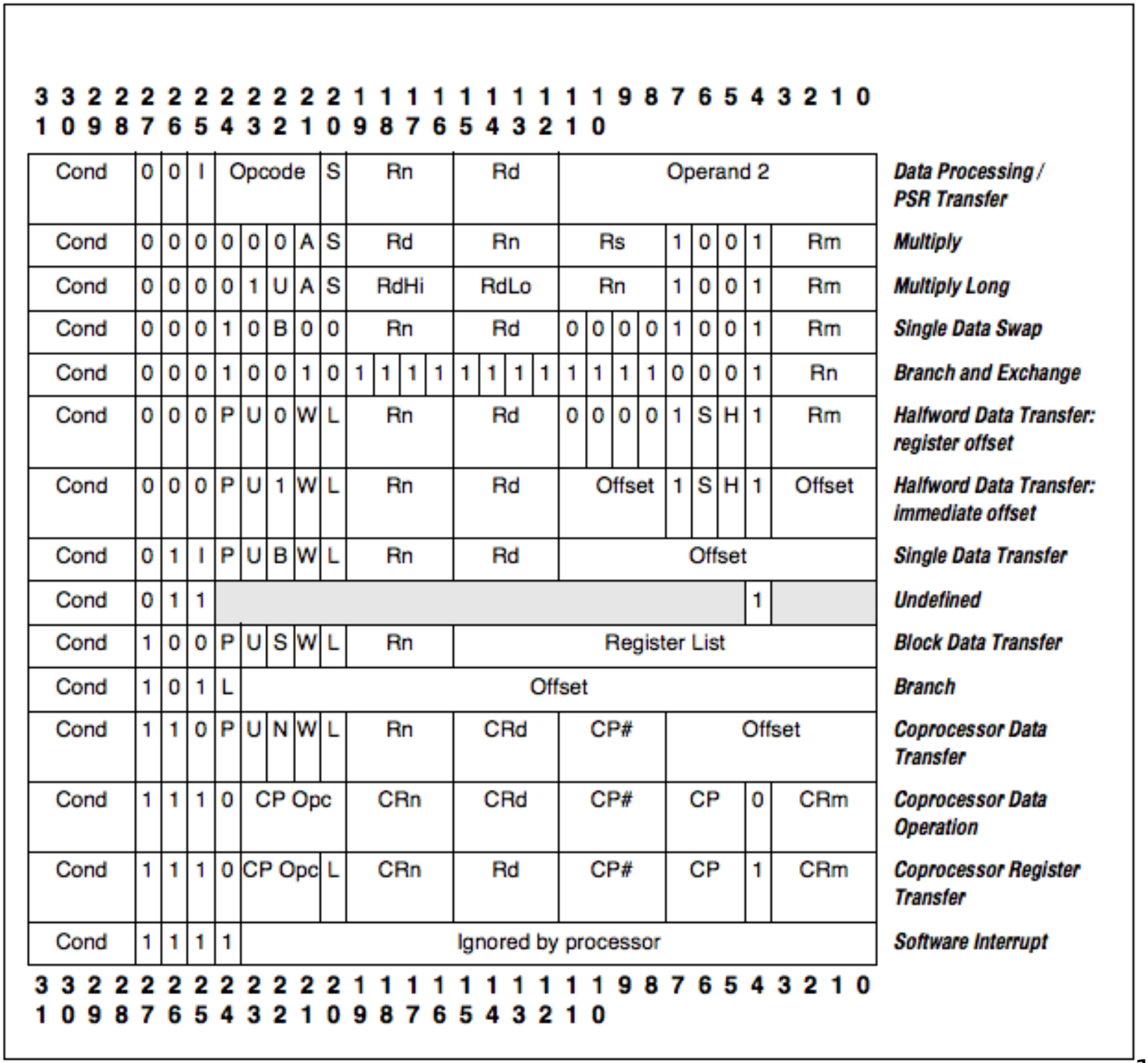


Figure 4-1: ARM instruction set formats

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

המידע הזה תופס מקום

- Large number of registers:

  \+ Enables better **register allocation** (and optimizations) by compiler

  \+ Fewer saves/restores

  \- Larger instruction size

  \- Larger register file size (יותר הספק)

השקף הבא מדגים את האתגר ב- register allocation

st = store
ld = load

דוגמה — מה יעשה הקומפיילר?

נניח שאנו עובדים עם 8 רגיסטרים
אם יש לנו 9 משתנים, למשל מטיפוס
int. חסר לנו מקום למשתנה אחד.

st z ("spilling") <==> push z
ld z (filling back) pop z

ישמור את אחד המשתנים במחסנית
כאשר מתפנה רגיסטר

"שפיכה" לזיכרון = spilling

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, auto-increment, auto-decrement, …)
- More addressing modes: (→ smaller semantic gap)

  + 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*

הסבר נוסף

Immediate (or literal)

ex: `ADD r1, r2, 1`
means: `r1 ← r2 + 1`
comment:used when a constant is needed.

Direct
ex: `ADD r1, r2, (100)`
means: `r1 ← r2 + M[100]`
comment: used to access static data; the address of the operand is include in the instruction; space must be provided to accommodate a whole address.

Register indirect (or register deferred)

ex: `ADD r1, r2, (r3)`
means: `r1 ← r2 + M[r3]`
comment: the register (r3 in this example) contains the address of a memory location.

Memory indirect (or memory deferred)

ex: `ADD r1, r2, @r3`
means: `r1 ← r2 + M[M[r3]]`
comment: used in pointer addressing; if r3 contains the address of a pointer p, then M[M[r3]] yields *p.

Other Examples of ISA-level Tradeoffs – נושאים מתקדמים

- Condition codes vs. not – פעולה על סמך קיומו של תנאי, see "Status Register"

VLIW (=Very Long Instruction Word) 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)
- …

בדיקת תלות ע"י החומרה או התוכנה

Back to *Programmer* vs. *(Micro)architect*

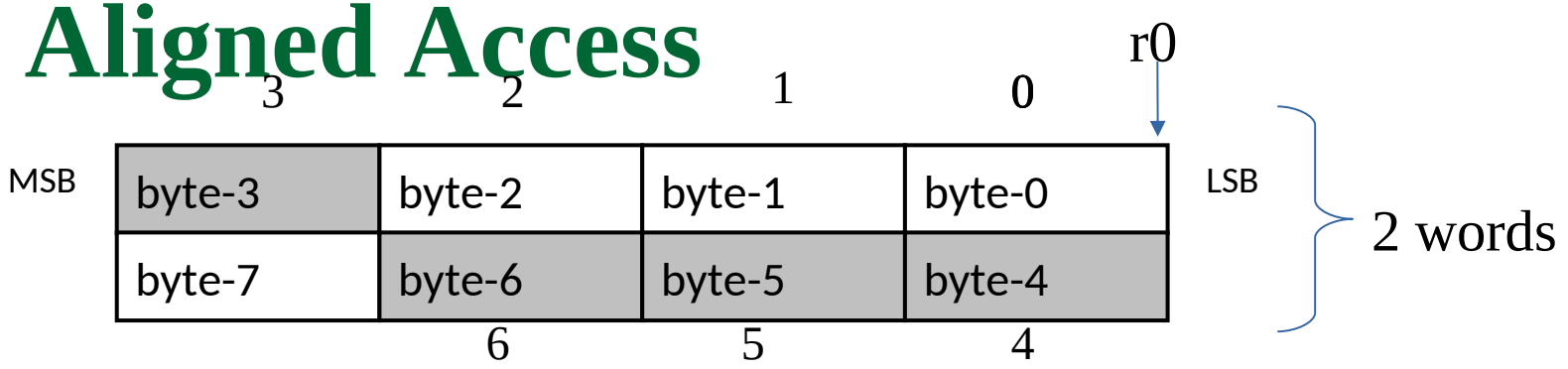
- Many ISA features designed to aid programmers
- But, complicate the hardware designer's job

דוגמאות לכך:

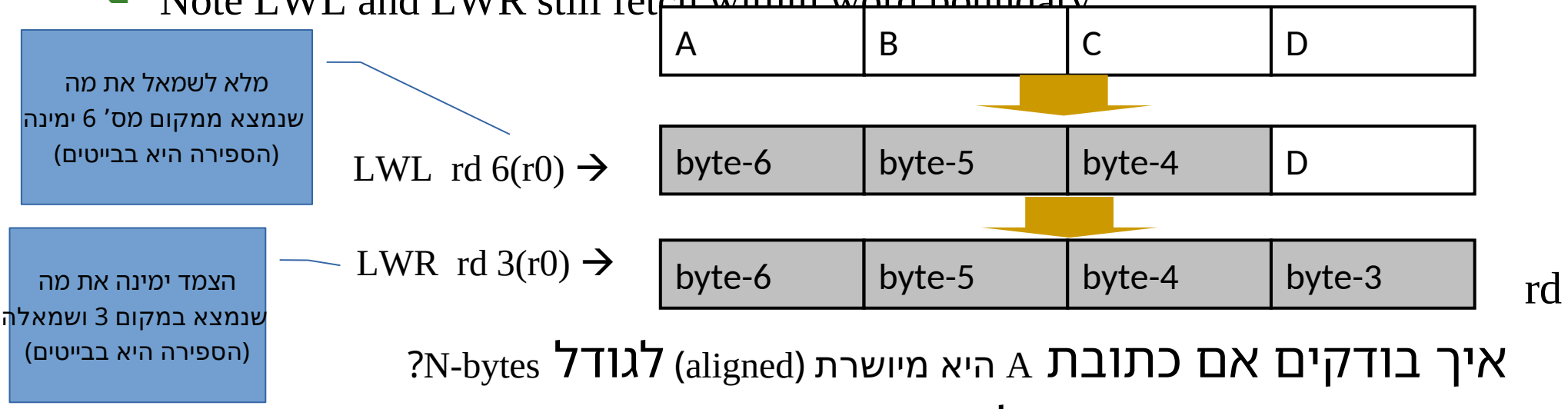
- Virtual memory
  - vs. overlay programming
  - Should the programmer be concerned about the size of code blocks fitting physical memory?
- Addressing modes
- Unaligned memory access
  - Compiler/programmer needs to align data

במערכות Embedded, כאשר תכנית גדולה לא נכנסת לזיכרון המוגבל ואז היא מחולקת למודולים שנטענים לזיכרון עפ"י הצורך. כמו-כן נעשה Swap עם מקטעים של תכניות אחרות

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 (=Load Word Left/Right) is slower
  - Note LWL and LWR still fetch within word boundary



איך בודקים אם כתובת A היא מיושרת (aligned) לגודל N-bytes?

פשוט בודקים את המודול A%N==0? עבור MIPS: N=4

Demo

In the next slide there is a screen shot of the MARS MIPS simulator

For the instructor:

working directory:

/home/telzur/science/Teaching/CPU/lectures/02/code

Code: lwl_lwr_demo.asm

Open mars in the terminal.

```
1   .data
2   number: .word 256 ישמש ככתובת
3   .text
4           addi $t0,$zero,123456
5           la $t1,number
6           sw $t0,0($t1)
7           lw $t2,0($t1)
8           lwl $t3, 0($t1)
9           lwr $t4, 0($t1)
10          lwl $t5, 1($t1)
11          lwr $t6, 1($t1)
12          lwl $t7, 2($t1)
13          lwr $s0, 2($t1)
14          lwl $s1, 3($t1)
15          lwr $s2, 3($t1)
16
```

המקום בבייטים

$$123456_{10} = 0001E240_{16}$$

	3	2	1	0
$t3	11	0x40000000		
$t4	12	0x0001e240		
$t5	13	0xe2400000		
$t6	14	0x000001e2		
$t7	15	0x01e24000		
$s0	16	0x00000001		
$s1	17	0x0001e240		
$s2	18	0x00000000		

טבלת ערכי הרגיסטרים בהגדלה

Edit | Execute

Text Segment

Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c010001	lui $1,0x00000001	4: addi $t0,$zero,123456 # initialize an arbitrary number = 0x0001E240, 123456(10)=1E240(16)
	0x00400004	0x3421e240	ori $1,$1,0x0000e240	
	0x00400008	0x00014020	add $8,$0,$1	
	0x0040000c	0x3c011001	lui $1,0x00001001	5: la $t1,number # set a memory location
	0x00400010	0x34290000	ori $9,$1,0x00000000	
	0x00400014	0xad280000	sw $8,0x00000000($9)	6: sw $t0,0($t1) # store the number in that location
	0x00400018	0x8d2a0000	lw $10,0x00000000($9)	7: lw $t2,0($t1) # load it again to verify
	0x0040001c	0x892b0000	lwl $11,0x00000000($9)	8: lwl $t3, 0($t1) #
	0x00400020	0x992c0000	lwr $12,0x00000000($9)	9: lwr $t4, 0($t1) #
	0x00400024	0x892b0001	lwl $11,0x00000001($9)	10: lwl $t3, 1($t1) #
	0x00400028	0x992c0001	lwr $12,0x00000001($9)	11: lwr $t4, 1($t1) #
	0x0040002c	0x892d0002	lwl $13,0x00000002($9)	12: lwl $t5, 2($t1) #
	0x00400030	0x992e0002	lwr $14,0x00000002($9)	13: lwr $t6, 2($t1) #
	0x00400034	0x892f0003	lwl $15,0x00000003($9)	14: lwl $t7, 3($t1) #
	0x00400038	0x99380003	lwr $24,0x00000003($9)	15: lwr $t8, 3($t1) #

Registers | Coproc 1 | Coproc 0

Name	Number	Value
$zero	0	0x00000000
$at	1	0x10010000
$v0	2	0x00000000
$v1	3	0x00000000
$a0	4	0x00000000
$a1	5	0x00000000
$a2	6	0x00000000
$a3	7	0x00000000
$t0	8	0x0001e240
$t1	9	0x10010000
$t2	10	0x0001e240
$t3	11	0xe2400000
$t4	12	0x000001e2
$t5	13	0x01e24000
$t6	14	0x00000001
$t7	15	0x0001e240
$s0	16	0x00000000
$s1	17	0x00000000
$s2	18	0x00000000
$s3	19	0x00000000
$s4	20	0x00000000
$s5	21	0x00000000
$s6	22	0x00000000
$s7	23	0x00000000
$t8	24	0x00000000

Data Segment

X86: Unaligned Access

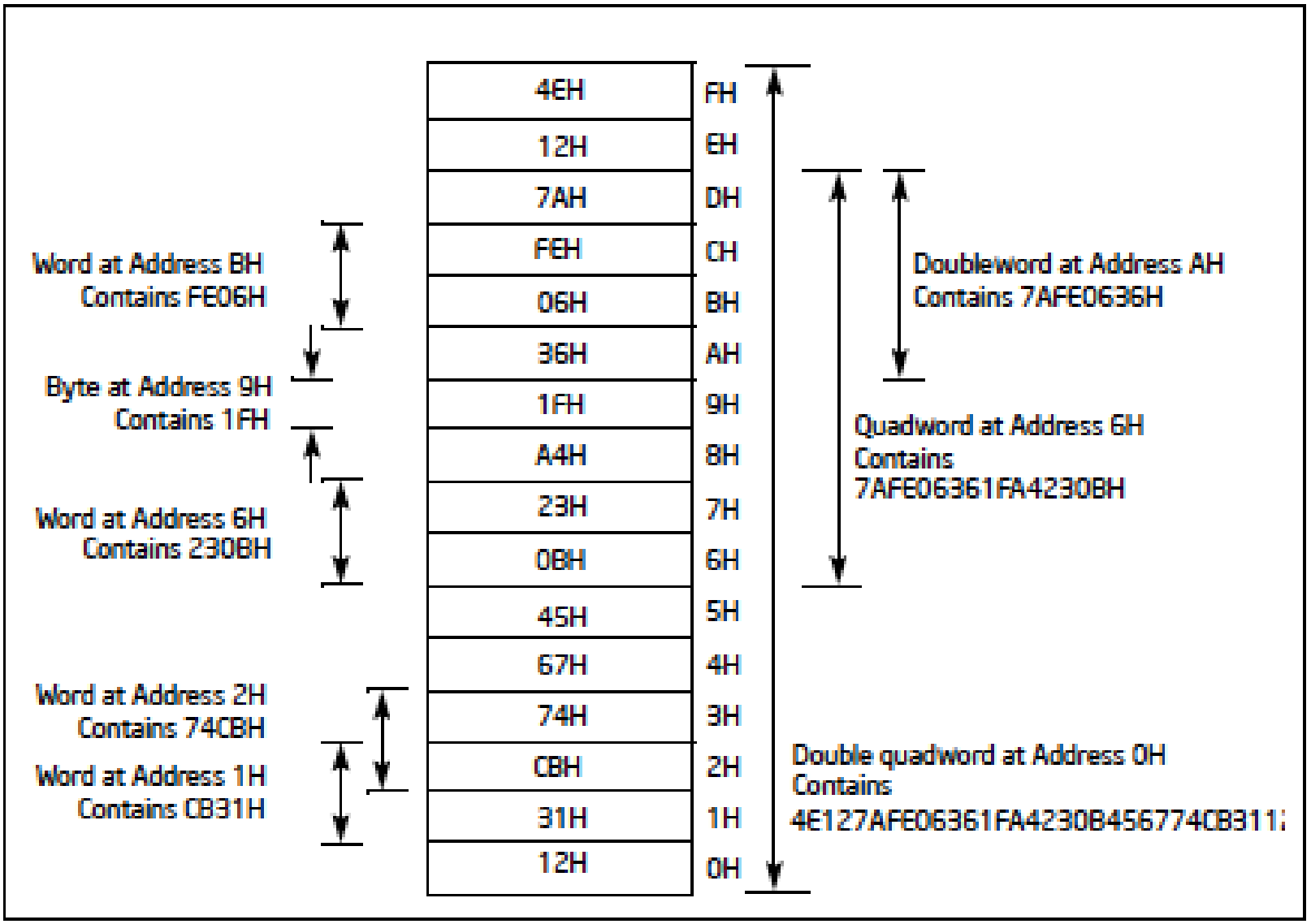


Figure 4-2. Bytes, Words, Doublewords, Quadwords, and Double Quadwords in Memory

גישה למילים
לא מיושרות

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.

יש תמיכה בכל מצב אבל יש תשלום בביצועים

What About ARM?

https://www.scss.tcd.ie/~waldroj/3d1/arm_arm.pdf

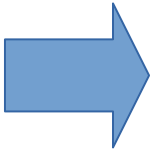
- ARM Architecture Reference Manual, Section A2.8

A2.8 Unaligned access support

The ARM architecture traditionally expects all memory accesses to be suitably aligned. In particular, the address used for a halfword access should normally be halfword-aligned, the address used for a word access should normally be word-aligned.

Prior to ARMv6, doubleword (LDRD/STRD) accesses to memory, where the address is not doubleword-aligned, are UNPREDICTABLE. Also, data accesses to non-aligned word and halfword data are treated as aligned from the memory interface perspective. That is:

- the address is treated as truncated, with address bits[1:0] treated as zero for word accesses, and address bit[0] treated as zero for halfword accesses.
- load single word ARM instructions are architecturally defined to rotate right the word-aligned data transferred by a non word-aligned address one, two or three bytes depending on the value of the two least significant address bits.
- alignment checking is defined for implementations supporting a System Control coprocessor using the A bit in CP15 register 1. When this bit is set, a Data Abort indicating an alignment fault is reported for unaligned accesses.



ARMv6 introduces unaligned word and halfword load and store data access support. When this is enabled, the processor uses one or more memory accesses to generate the required transfer of adjacent bytes transparently to the programmer, apart from a potential access time penalty where the transaction crosses an IMPLEMENTATION DEFINED cache-line, bus-width or page boundary condition. Doubleword accesses must be word-aligned in this configuration.

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…

RISC vs. CISC

RISC	CISC
Multiple register sets, often consisting of more than 256 registers	Single register set, typically 6 to 16 registers total
Three register operands allowed per instruction (e.g., add R1, R2, R3)	One or two register operands allowed per instruction (e.g., add R1, R2)
Parameter passing through efficient on-chip register windows	Parameter passing through inefficient off-chip memory
Single-cycle instructions (except for load and store	Multiple-cycle instructions
Hardwired control	Microprogrammed control
Highly pipelined	Less pipelined
Simple instructions that are few in number	Many complex instructions
Fixed-length instructions	Variable-length instructions
Complexity in compiler	Complexity in microcode
Only load and store instructions can access memory	Many instructions can access memory
Few addressing modes	Many addressing modes

Credit:
"The Essentials of Computer Organization and Architecture", 5th edition by Linda Null

עד כאן הצגת נושא ה- ISA באופן כללי.
מכאן נמשיך ונכיר את ISA של מעבד MIPS

MIPS ISA

במצגת הבאה נלמד בפירוט
על ה-ISA של מעבד MIPS.
בהמשך מצגת זו שקפים אודות ה-ISA
של MIPS במבט-על

חומר הלימוד על אסמבלי של **MIPS (המצגת הבאה)** נלקח מתוך פרק **6** בספר של **H&H**. המצגות נמצאות במודל.

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

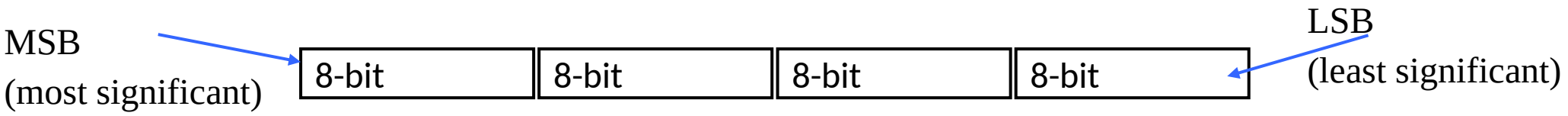
Data Format

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

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 . . . . . .

MSB Big Endian 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

MSB Little Endian 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?

htonl = Host TO Network Long

It's used in network programming to convert a 32-bit integer from host byte order to network byte order (big-endian). This is important because different systems may use different byte orders, and network protocols typically use big-endian order.

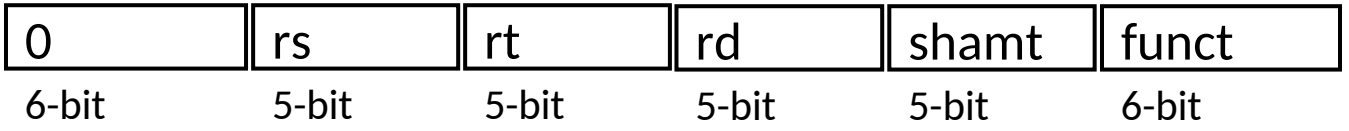
```
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$ cat ./test_htonl2.c
#include <stdio.h>
#include <arpa/inet.h>

int main() {
    unsigned long hostlong = 0x12345678;
    unsigned long netlong = htonl(hostlong);
    printf("Host order: 0x%lx\n", hostlong);
    printf("Network order: 0x%lx\n", netlong);
    return 0;
}
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$ ./test_htonl2
Host order: 0x12345678
Network order: 0x78563412
telzur@TUF:~/science/Teaching/CPU/lectures/02/code$
```

My computer byte order is Little Endian
and this is why the order got reversed

Instruction Formats

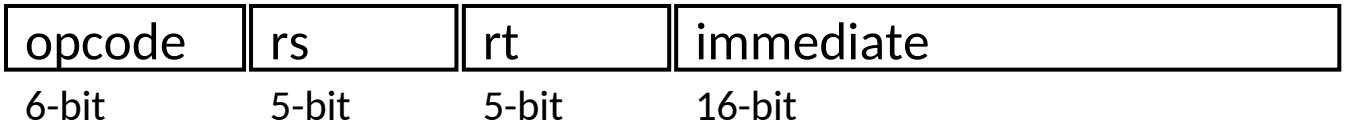
- 3 simple formats

  R-type, 3 register operands



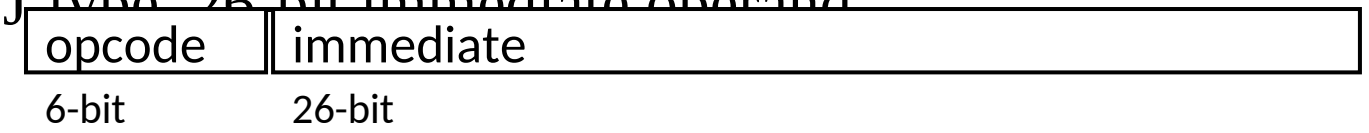
R-type

  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)

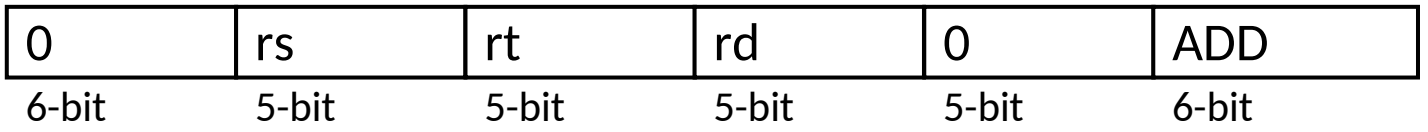
  format and fields readily extractable

ALU Instructions

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

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

- Machine encoding



R-type

- 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

3 ביט → 8 מצבים

5...3 \ 2...0	0	1	2	3	4	5	6	7
	SPECIAL function							
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]

ההוראות היסודיות (סה"כ 6 ביט)

הוראות מיוחדות – הרחבת ה-ISA – ראו השקף הבא.

What patterns do you see? Why are they there?

הרחבה של ה–ISA ב–MIPS R4000

Table 1-16 Extensions to the ISA: Exception Instructions

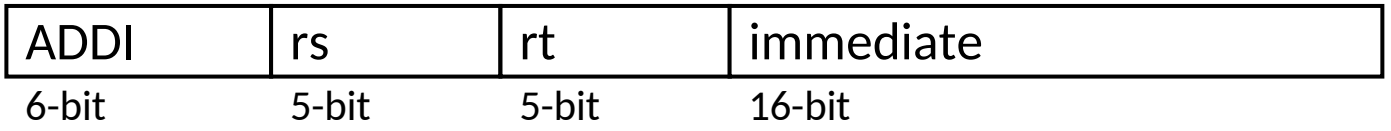
OpCode	Description
TGE	Trap if Greater Than or Equal
TGEU	Trap if Greater Than or Equal Unsigned
TLT	Trap if Less Than
TLTU	Trap if Less Than Unsigned
TEQ	Trap if Equal
TNE	Trap if Not Equal
TGEI	Trap if Greater Than or Equal Immediate
TGEIU	Trap if Greater Than or Equal Immediate Unsigned
TLTI	Trap if Less Than Immediate
TLTIU	Trap if Less Than Immediate Unsigned
TEQI	Trap if Equal Immediate
TNEI	Trap if Not Equal Immediate

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}

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

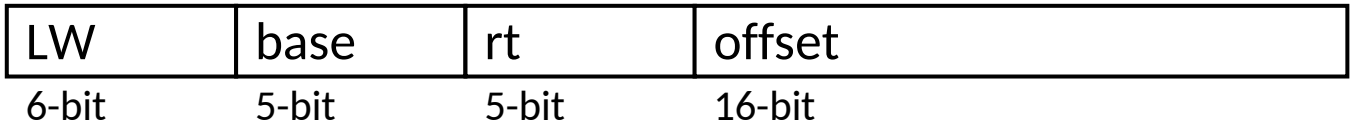
```
add r_temp r_g r_h   # r_temp = g+h
add r_f r_i r_j      # r_f = i+j
sub r_f r_temp r_f   # f = r_temp – r_f
```

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?

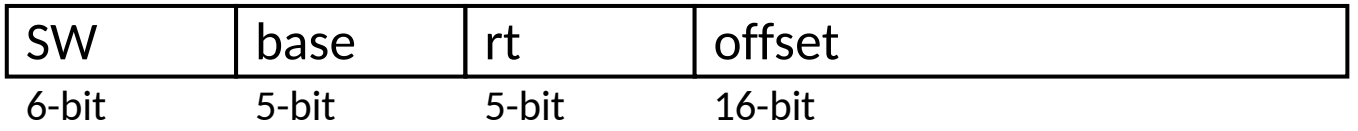
  MMU (=Memory Management Unit) exceptions

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)

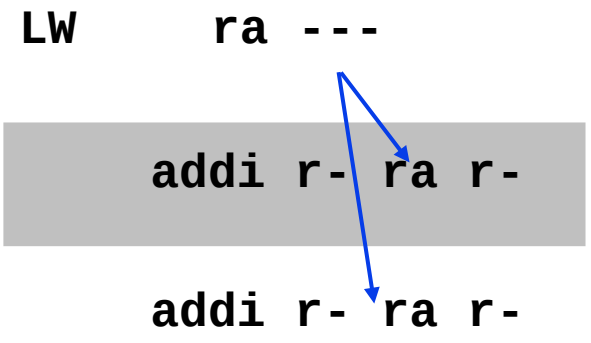
- Assembly Code

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)

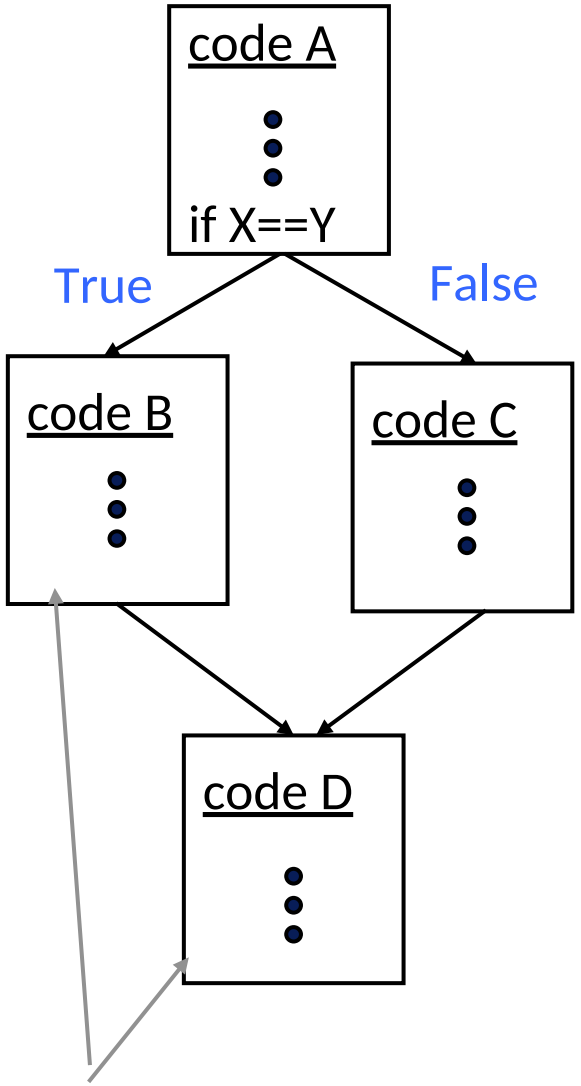
*BTW, notice that latency is defined in “instructions” not cyc. or sec.

Control Flow Instructions

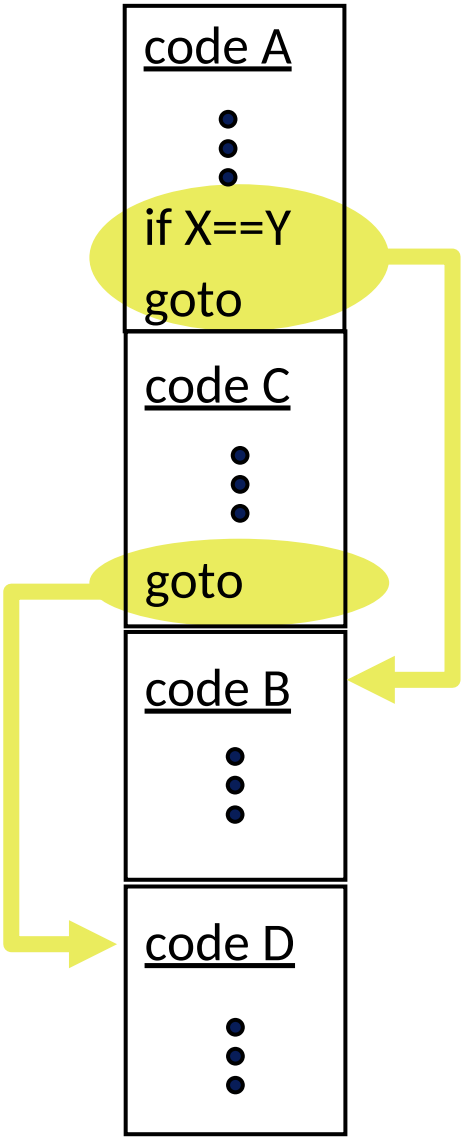
- C-Code

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

Control Flow Graph



Assembly Code
(linearized)



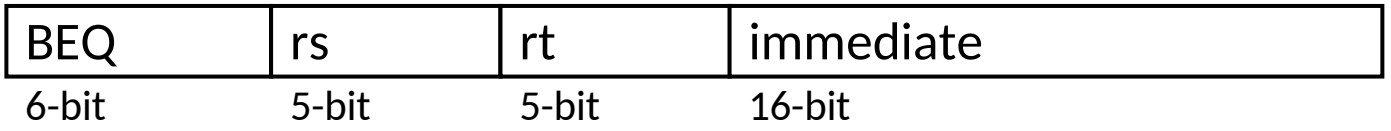
these things are called basic blocks

(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

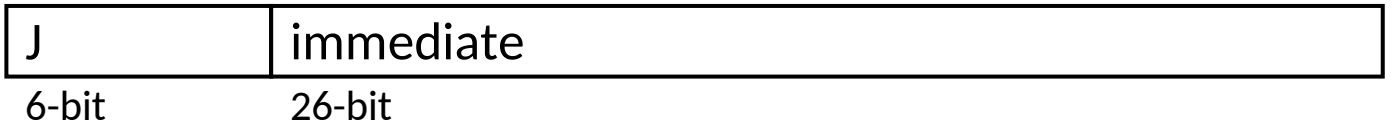
Why isn't there a BLE or BGT instruction?

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

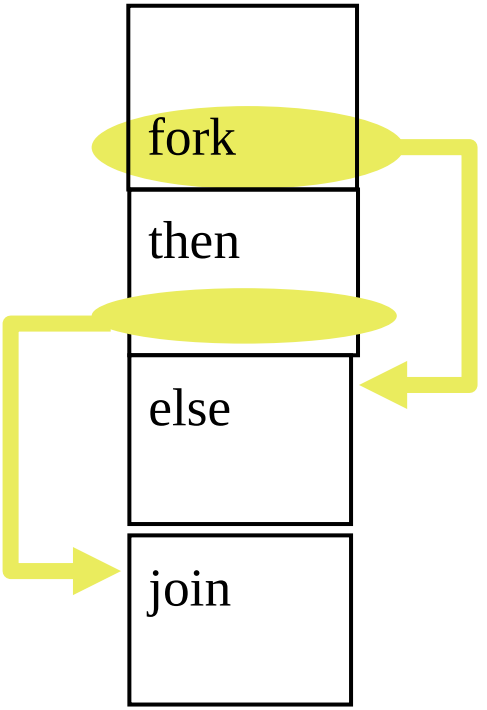
  - Jump and Link
  - Jump Registers

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
```

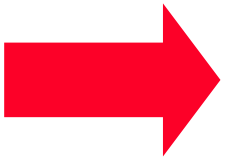
Branch Delay Slots

- 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
          . . . .
```

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**  $_{\text{call}}$ **B**  $_{\text{return}}$ **C**  $_{\text{call}}$ **B**  $_{\text{return}}$ **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

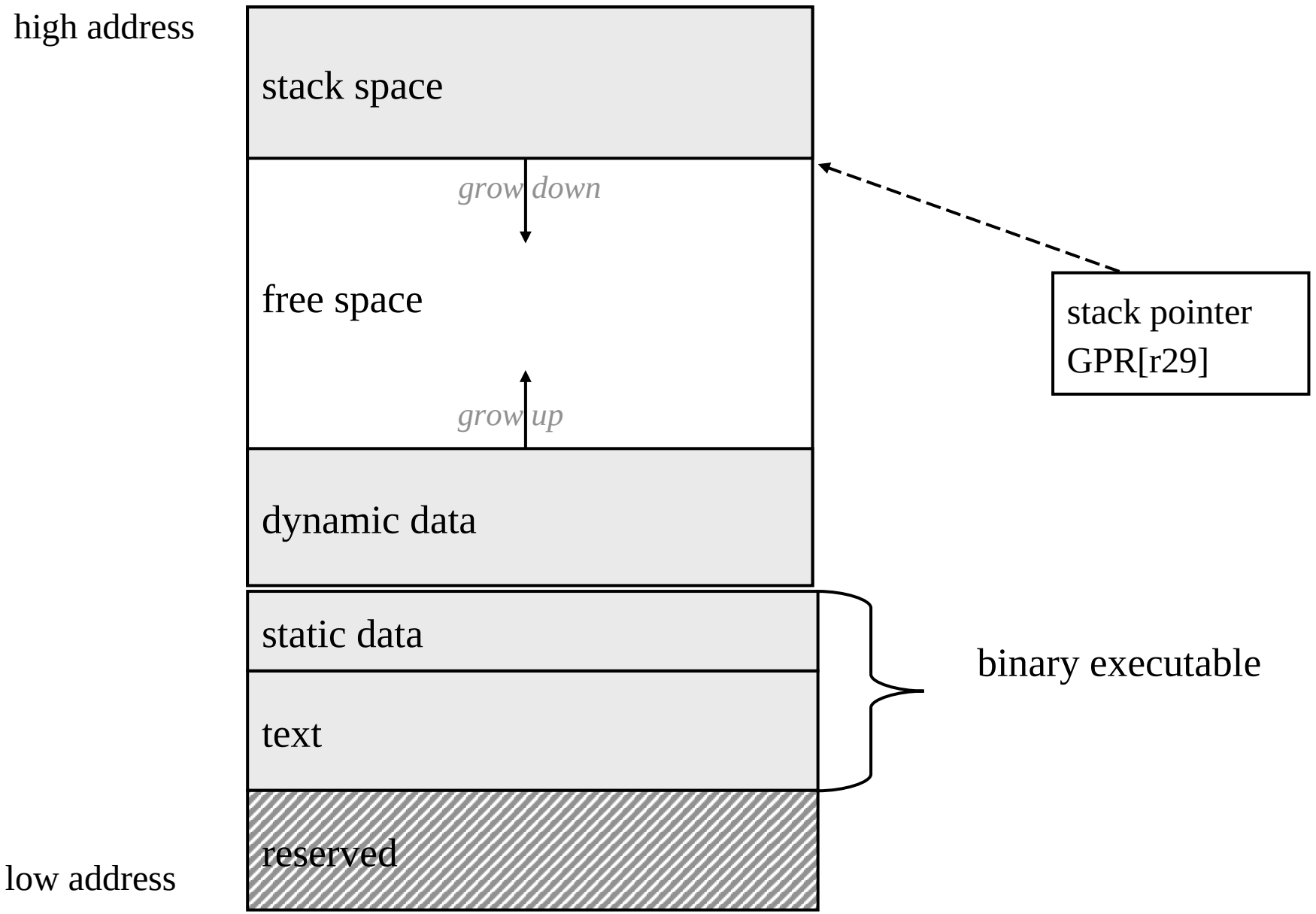
  Caller says to callee, “If there is anything I care about in these registers, I already saved it myself.”

  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) .....

1. callee loads results to r2, r3
2. callee restores saved register values
3. JR r31
4. caller continues with return values in r2, r3

epilogue

.....

To Summarize: RISC-V / 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