

Chapter 2

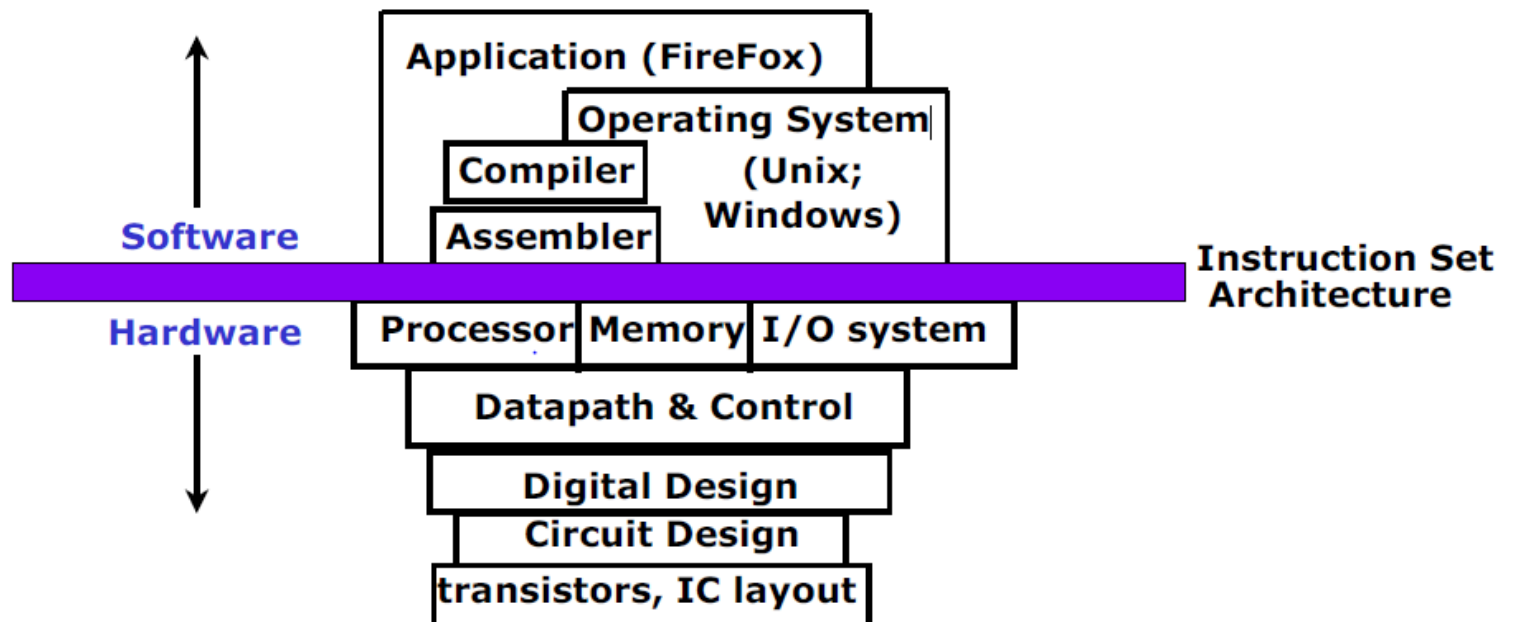
Instructions: Language of the Computer

Instruction Set

- **Instruction:** The words of a computer's language
- **Instruction set:** The vocabulary of commands understood by a given architecture.
- Different computers have different instruction sets
 - But with many aspects in common
 - Example: basic operations
- Early computers had very simple instruction sets
 - Simplified implementation
- Many modern computers also have simple instruction sets
 - Intel x86 powers the cloud of the post-PC era

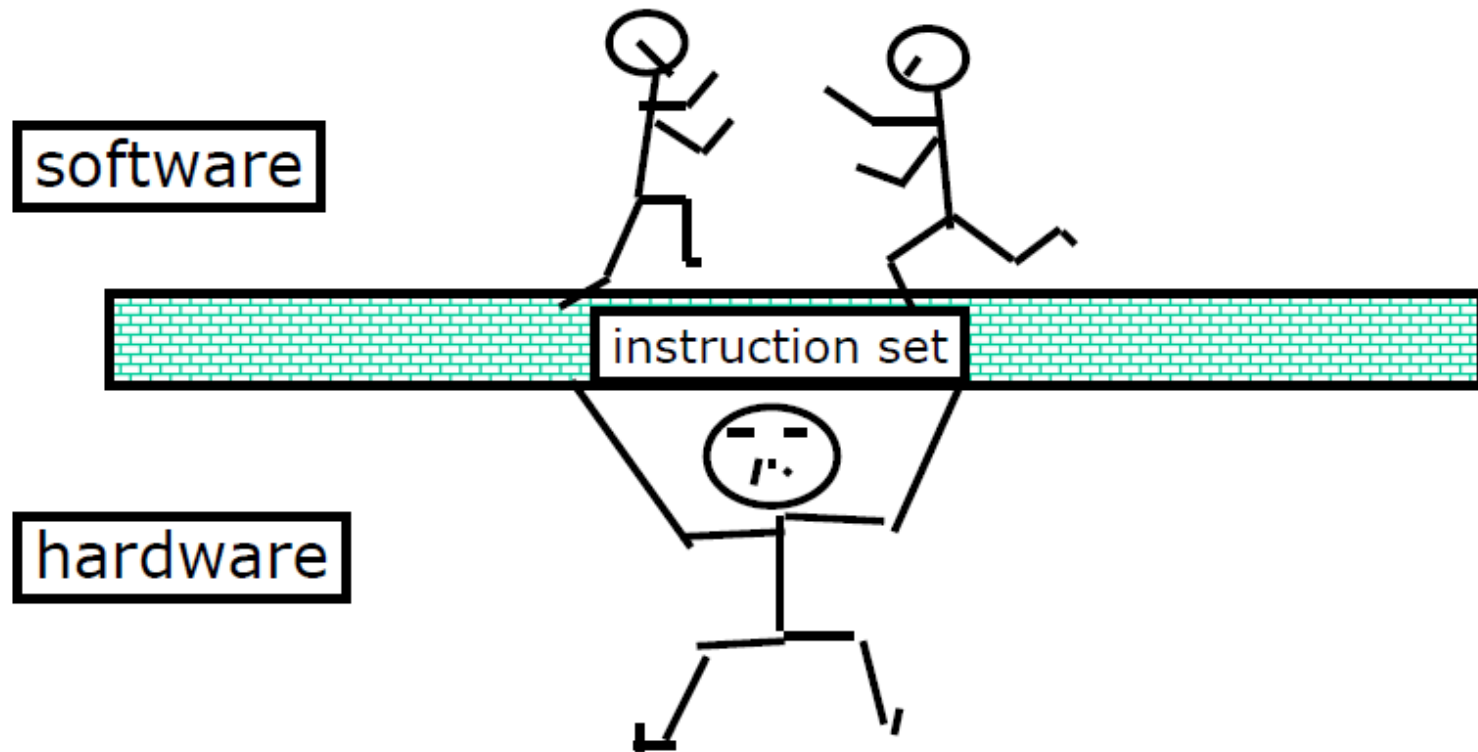
Instruction Set Architecture (ISA)

- Key Idea: abstraction
 - hide unnecessary implementation details
 - helps us cope with enormous complexity of real systems



The Instruction Set: a Critical Interface

The actual programmer visible hardware view



Classifications

- **Complex instruction set computer (CISC)**
 - has many specialized instructions, some of which may only be rarely used in practical programs
- **Reduced instruction set computer (RISC)**
 - Implementing only the instructions that are frequently used in programs, while the less common operations are implemented as subroutines by software (generated by compiler)
- **RISC have fewer cycles per instruction (CPI) than a CISC**

Popular ISA

Name	CISC/RISC	Word size	applications	comments
X86/IA32 X86-64/X64	CISC	32/64	desktops, laptops to servers	Dominant
ARM ARMV8-A	RISC	32/64	Mobile, embedded	Surpassing x86 in numbers; licensing IP
MIPS/MIPS32 MIPS64	RISC	32/64	Embedded, network	Classical RISC ISA; licensing IP; struggling
PowerPC/Power	RISC	32/64	IBM, Mac PC, HPC	Dying ISA
SPARC	RISC	32/64	Sun Micro System, servers	Dying ISA
RISC-V	RISC	32/64/128	Embedded	Open source, rising ISA

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
 - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
 - Applications in consumer electronics, network/storage equipment, cameras, printers, ...

Arithmetic Operations

- Add and subtract, three operands
 - Two sources and one destination

add a, b, c // a gets b + c
- All arithmetic operations have this form
- *Design Principle 1*: Simplicity favours regularity
 - Regularity makes implementation simpler
 - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

`a = b + c;`

`d = a - e;`

- Compiled RISC-V code:

`add a , b, c`

`sub d, a, e`

Arithmetic Example

- C code:

$f = (g + h) - (i + j);$

- Compiled RISC-V code:

```
add t0, g, h    // temp t0 = g + h
add t1, i, j    // temp t1 = i + j
sub f, t0, t1    // f = t0 - t1
```

Registers

- Once a memory is fetched, the data must be placed somewhere in CPU
- Registers are a small set of fast memory frequently used by CPU during execution of instructions
- Processor registers are normally at the top of the memory hierarchy and provide the fastest way to access data.
 - A CPU register can generally be accessed in a single clock cycle, whereas main memory may require dozens of CPU clock cycles to read or write.
- registers can hold variables and intermediate results
- Many instructions use registers as operands

Registers

- code for $A = B + C$ (This is not RISC-V, It is in English)
 - load R1,B # R1 = B
 - load R2,C # R2 = C
 - add R3,R1,R2 # R3 = R1+R2
 - store R3,A # A = R3
- Early ISAs supported a few registers (8 or less) (Intel's X86)
- Many current processors support 32 registers (RISC-V)
- The more registers available, the fewer memory accesses will be necessary
 - Registers can hold lots of intermediate values
- Instructions must include bits to specify which registers to operate on
 - register address

Register Operands

- RISC-V has a 32×32 -bit register file
 - Use for frequently accessed data
 - 32-bit data is called a “word”
- 1 special purpose register
 - program counter (PC) - holds the address of the current instruction.
- *Design Principle 2: Smaller is faster*
 - c.f. main memory: millions of locations

RISC-V Registers

x+ # of the register in instructions

- x0: the constant value 0
- x1: return address
- x2: stack pointer
- x3: global pointer
- x4: thread pointer
- x5 – x7, x28 – x31: temporaries
- x8: frame pointer
- x9, x18 – x27: saved registers
- x10 – x11: function arguments/results
- x12 – x17: function arguments

Register Operand Example

- C code:

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

- `f, g, ..., j` in `x19, x20, ..., x23`

- Compiled RISC-V code:

`add x5, x20, x21`

`add x6, x22, x23`

`sub x19, x5, x6`

Memory Operands

- Main memory used for composite data
 - Arrays, structures, dynamic data
- To apply arithmetic operations
 - Load values from memory into registers
 - Store result from register to memory
- Memory is byte addressed
 - Each address identifies an 8-bit byte
- RISC-V is Little Endian
 - Least-significant byte at least address of a word
 - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory
 - Unlike some other ISAs

Memory as Array

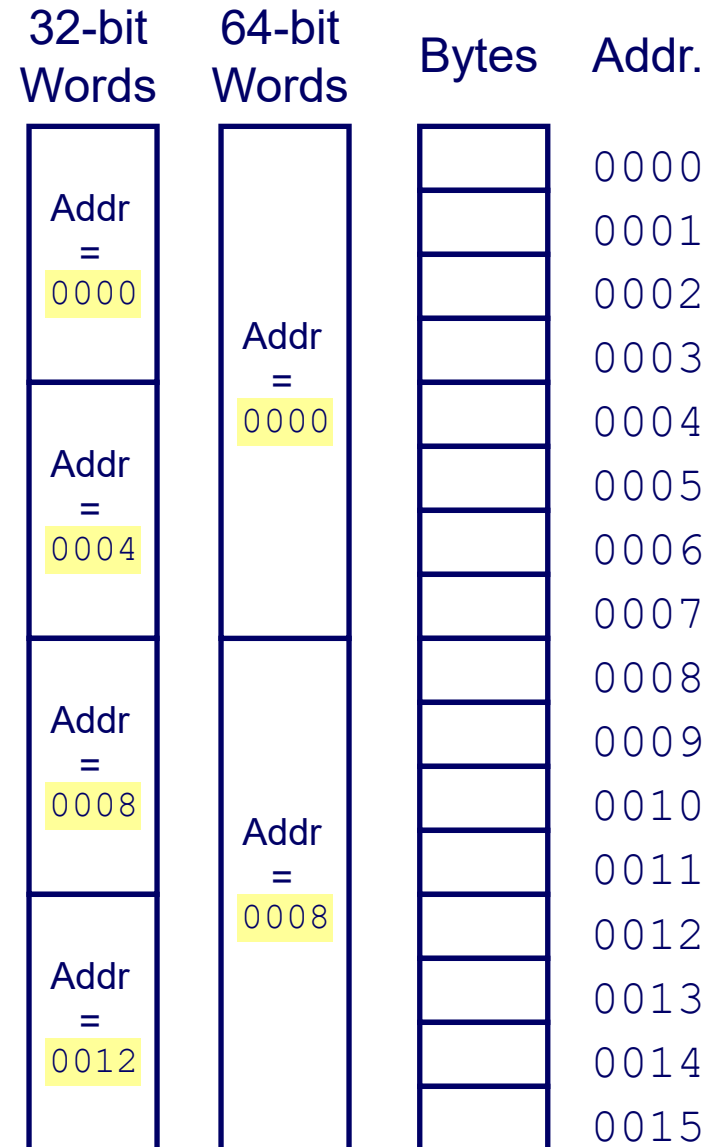
Computer		Programmers		
Address	Content	Name	Type	Value
90000000	00	sum	int (4 bytes)	000000FF (255 ₁₀)
90000001	00			
90000002	00			
90000003	FF			
90000004	FF	age	short (2 bytes)	FFFF (-1 ₁₀)
90000005	FF			
90000006	1F	average	double (8 bytes)	1FFFFFFFFFFFFFFFFF (4.45015E-308 ₁₀)
90000007	FF			
90000008	FF			
90000009	FF			
9000000A	FF			
9000000B	FF			
9000000C	FF			
9000000D	FF			
9000000E	90	ptrSum	int* (4 bytes)	90000000
9000000F	00			
90000010	00			
90000011	00			

Note: All numbers in hexadecimal

Word-Oriented Memory Organization

■ Addresses Specify Byte Locations

- Which is the address of the first byte in word.
- Addresses of successive words differ by 4 (32-bit) or 8 (64-bit).
- Addresses of multi-byte data items are typically **aligned** according to the size of the data.
- Otherwise, it is called **unaligned access**



Byte Ordering Examples

- **Big Endian:** Most significant byte has lowest (first) address.
- **Little Endian:** Least significant byte has lowest address
- **Example:**
 - Int variable x has 4-byte representation `0x01234567`
 - Most significant byte is `0x01`;
 - Least significant byte is `0x67`
 - Address given by `&x` is `0x100`

Big Endian

Address:			0x100	0x101	0x102	0x103	
Value:			01	23	45	67	

Little Endian

Address:			0x100	0x101	0x102	0x103	
Value:			67	45	23	01	

Memory Operand Example

- C code:

`A[12] = h + A[8];`

- `h` in `x21`, base address of `A` in `x22`

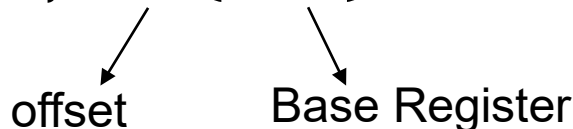
- Compiled RISC-V code:

- Index 8 requires offset of 32

- 4 bytes per word

```
lw      x9, 32(x22)
add     x20, x21, x9
sw      x20, 48(x22)
```

offset Base Register



Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
 - More instructions to be executed
- Compiler must use registers for variables as much as possible
 - Only spill to memory for less frequently used variables
 - Register optimization is important!

Constant or Immediate Operands

- Constant data specified in an instruction

`addi x22, x22, 4` // $x22 = x22 + 4$

- Make the common case fast

- Small constants are common

- More than half of the arithmetic instructions have a constant

- Immediate operand avoids a load instruction

`lw x9, AddrConstant4(x3)`

`add x22, x22, x9`

The Constant Zero

- register x0 (\$zero) is the constant 0
 - Cannot be overwritten
 - It is legal to update \$zero, but not any effect
 - "add \$zero, \$t1, \$t2" is basically killing one clock without doing anything
- Useful for some common operations
 - move between registers
add x20, x22, x0

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: 0 to $+2^n - 1$

- Example

- $0000\ 0000\ \dots\ 0000\ 1011_2$
 $= 0 + \dots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
 $= 0 + \dots + 8 + 0 + 2 + 1 = 11_{10}$

- Word (RISC-V) 32 bits: 0 to $+4,294,967,295_{10}$ ($2^{32} - 1$)

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \dots + x_12^1 + x_02^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$

- Example

- $1111\ 1111\ \dots\ 1111\ 1100_2$
 $= -1 \times 2^{31} + 1 \times 2^{30} + \dots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
 $= -2,147,483,648 + 2,147,483,644 = -4_{10}$

- Using 32 bits: $-2,147,483,648$ to $+2,147,483,647$

2s-Complement Signed Integers

- Bit 31 is sign bit
 - 1 for negative numbers
 - 0 for non-negative numbers
- $-(-2^n - 1)$ can't be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers
 - 0: 0000 0000 ... 0000
 - -1: 1111 1111 ... 1111
 - Most-negative: 1000 0000 ... 0000
 - Most-positive: 0111 1111 ... 1111

Signed Negation

- Complement and add 1
 - Complement means $1 \rightarrow 0, 0 \rightarrow 1$

$$x + \bar{x} = 1111 \dots 111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate $+2_{\text{ten}}$
 - $+2 = 0000 \ 0000 \ \dots \ 0010_{\text{two}}$
 - $-2 = 1111 \ 1111 \ \dots \ 1101_{\text{two}} + 1$
 $= 1111 \ 1111 \ \dots \ 1110_{\text{two}}$

Sign Extension

- Representing a number using more bits
 - Preserve the numeric value
- Replicate the sign bit to the left
 - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
 - +2: 0000 0010 => 0000 0000 0000 0010
 - -2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
 - 1b: sign-extend loaded byte
 - 1bu: zero-extend loaded byte

Hexadecimal

- Base 16
 - Compact representation of bit strings
 - 4 bits per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: eca8 6420
 - 1110 1100 1010 1000 0110 0100 0010 0000

- How to represent instructions in the computer?
- -- machine code

Representing Instructions

- Instructions are encoded in binary
 - Called **machine code**
- RISC-V instructions
 - Encoded as 32-bit instruction words
 - Small number of formats encoding operation code (opcode), register numbers, ...
 - Regularity!

RISC-V R-format Instructions



■ Instruction fields

- opcode: operation code
- rd: destination register number
- funct3: 3-bit function code (additional opcode)
- rs1: the first source register number
- rs2: the second source register number
- funct7: 7-bit function code (additional opcode)

R-format Example

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

add x9, x20, x21

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

Machine code :

0000 0001 0101 1010 0000 0100 1011 0011_{two} =
015A04B3₁₆

RISC-V I-format Instructions



- Immediate arithmetic and load instructions
 - rs1: source or base address register number
 - immediate: constant operand, or offset added to base address
 - 2s-complement value (-2^{11} to $2^{11}-1$), sign extended
- *Design Principle 3: Good design demands good compromises*
 - Different formats complicate decoding, but allow 32-bit instructions uniformly
 - Keep formats as similar as possible

I-format Example - addi

immediate	rs1	funct3	rd	opcode
12 bits	5 bits	3 bits	5 bits	7 bits

addi x1, x2, 1000

1000	2	0	1	19
------	---	---	---	----

001111101000	00010	000	00001	0010011
--------------	-------	-----	-------	---------

Machine code:

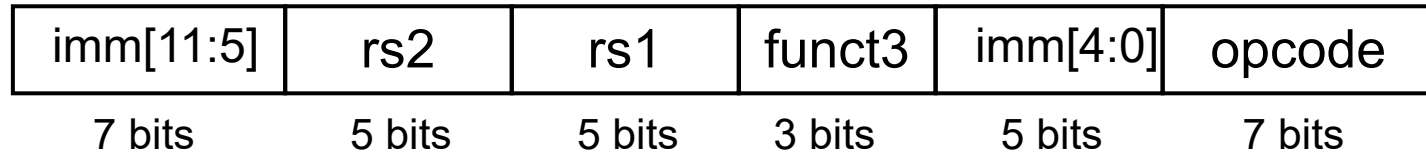
0011 1110 1000 0001 0000 0000 1001 0011₂

RISC-V S-format Instructions

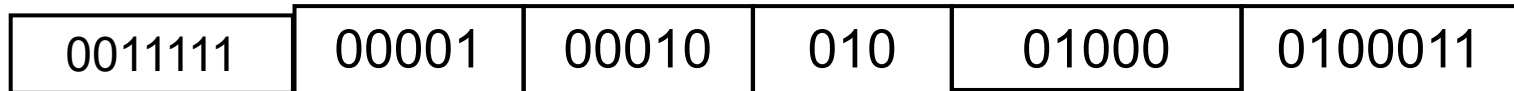


- Different immediate format for store instructions
 - rs1: base address register number
 - rs2: source operand register number
 - immediate: offset added to base address
 - Split so that rs1 and rs2 fields always in the same place

S-format Example - sw



sw x1, 1000(x2)

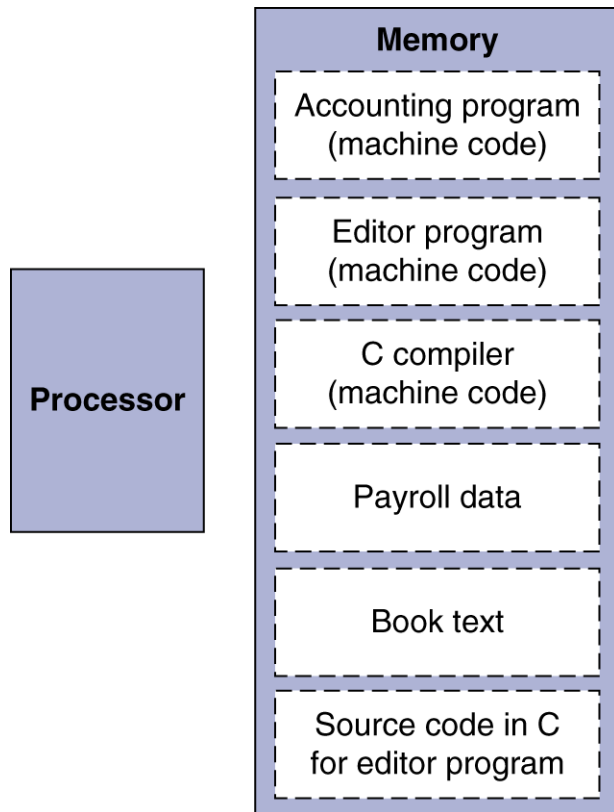


Machine code:

0011 1110 0001 0001 0010 0100 0010 0011₂

Stored Program Computers

The BIG Picture



- Instructions represented in binary, just like data
- Instructions and data stored in memory
- Programs can operate on programs
 - e.g., compilers, linkers, ...
- Binary compatibility allows compiled programs to work on different computers
 - Standardized ISAs

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	sll, slli
Shift right	>>	>>>	srl, srli
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR			or, ori
Bit-by-bit XOR	^	^	xor, xori
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

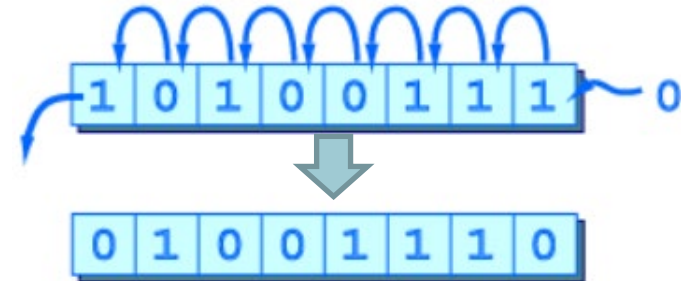
Shift Operations

funct6	immed	rs1	funct3	rd	opcode
6 bits	6 bits	5 bits	3 bits	5 bits	7 bits

- immed: how many positions to shift
- Shift left logical
 - Shift left and fill with 0 bits
 - $sll\ i$ by i bits multiplies by 2^i
- Shift right logical
 - Shift right and fill with 0 bits
 - $srl\ i$ by i bits divides by 2^i (unsigned only)

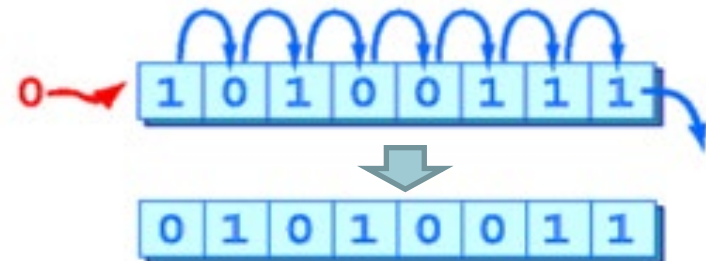
- Example: Shifting 1 bit to the left

- move all the bits to the left
- insert zero from the right.



- Shifting 1 bit to the right

- move all the bits to the right
- insert zero from the



AND Operations

- Leaves a 1 if both bits of the operands are 1
- Useful to mask bits in a word
 - Select some bits, clear others to 0

and x9, x10, x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00001100 00000000

OR Operations

- 1 if either operand bit is a 1
 - Useful to include bits in a word
 - Set some bits to 1, leave others unchanged
- or x9, x10, x11

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x11	00000000 00000000 00000000 00000000 00000000 00000000 00111100 00000000
x9	00000000 00000000 00000000 00000000 00000000 00000000 00111101 11000000

XOR Operations

- 0 when bits are the same, 1 if they are different
- Differencing operation
 - Set some bits to 1, leave others unchanged

`xor x9,x10,x12` // NOT operation

x10	00000000 00000000 00000000 00000000 00000000 00000000 00001101 11000000
x12	11111111 11111111 11111111 11111111 11111111 11111111 11111111 11111111
x9	11111111 11111111 11111111 11111111 11111111 11111111 11110010 00111111

Conditional Operations

- Branch to a labeled instruction if a condition is true
 - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
 - if (`rs1 == rs2`) branch to instruction labeled L1
- `bne rs1, rs2, L1`
 - if (`rs1 != rs2`) branch to instruction labeled L1

Compiling If Statements

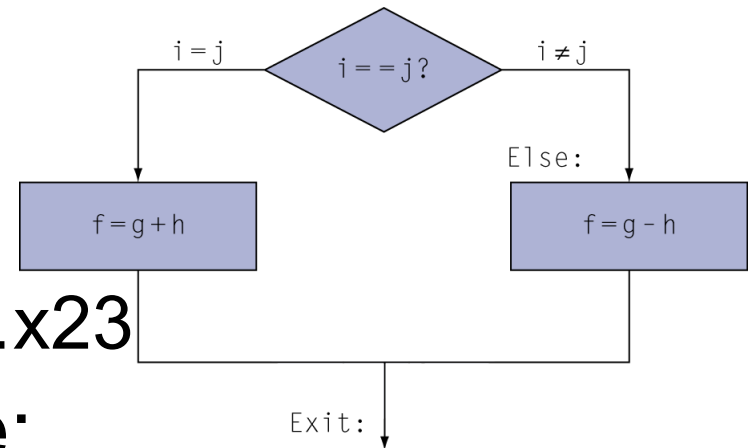
- C code:

```
if (i==j) f = g+h;  
else f = g-h;
```

- f, g, h, i, j in x19, x20, ...x23

- Compiled RISC-V code:

```
    bne x22, x23, Else  
    add x19, x20, x21  
    beq x0,x0,Exit // unconditional  
Else: sub x19, x20, x21  
Exit: ...
```



Assembler calculates addresses

Compiling Loop Statements

- C code:

```
while (save[i] == k) i += 1;
```

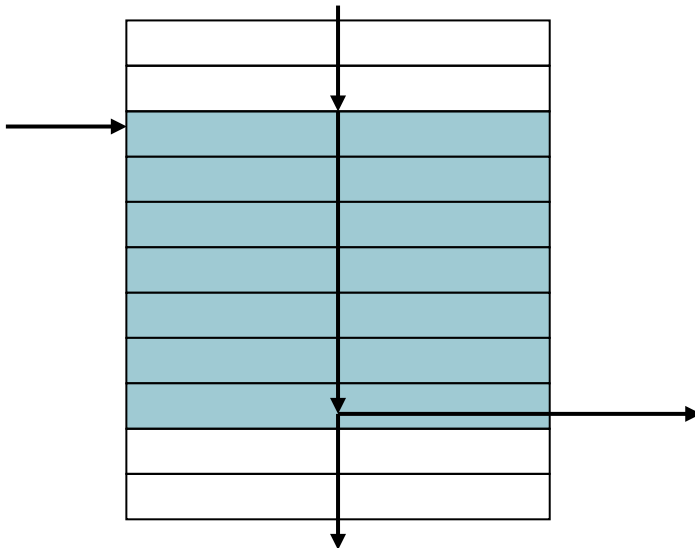
- i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 2    //temp reg x10=i*4
      add  x10, x10, x25  //x10 = address of save[i]
      ld   x9, 0(x10)     //temp reg x9 = save[i]
      bne  x9, x24, Exit  //go to Exit if save[i]!=k
      addi x22, x22, 1    // i = i + 1
      beq  x0, x0, Loop   // go to loop
Exit: ...
```

Basic Blocks

- A basic block is a sequence of instructions with
 - No embedded branches (except at end)
 - No branch targets (except at beginning)



- A compiler identifies basic blocks for optimization
- An advanced processor can accelerate execution of basic blocks

More Conditional Operations

- `blt rs1, rs2, L1`
 - if ($rs1 < rs2$) branch to instruction labeled L1
- `bge rs1, rs2, L1`
 - if ($rs1 \geq rs2$) branch to instruction labeled L1
- Example
 - if ($a > b$) $a += 1$;
 - a in `x22`, b in `x23`
`bge x23, x22, Exit` // branch if $b \geq a$
`addi x22, x22, 1`

Exit:

Signed vs. Unsigned

- Signed comparison: blt, bge
- Unsigned comparison: bltu, bgeu
- Example
 - $x22 = 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111$
 - $x23 = 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001$
 - $x22 < x23$ // signed
 - $-1 < +1$
 - $x22 > x23$ // unsigned
 - $+4,294,967,295 > +1$

Procedure Calling

- Or function: perform a specific task
- Steps required
 1. Place parameters in registers x10 to x17
 2. Transfer control to procedure
 3. Acquire storage for procedure
 4. Perform procedure's operations
 5. Place result in register for **caller**
 6. Return to place of call (address in x1)

**Caller: The program that instigates a procedure*

Procedure Call Instructions

- Procedure call: jump and link
`jal x1, ProcedureLabel`
 - Address of following instruction put in x1
 - Jumps to target address
- Procedure return: jump and link register
`jalr x0, 0(x1)`
 - Like jal, but jumps to 0 + address in x1
 - Use x0 as rd (x0 cannot be changed)
 - Can also be used for computed jumps
 - e.g., for case/switch statements

- Classification of Procedures
 - Leaf Procedure: do not call others
 - Non-Leaf Procedure: call others
 - recursive procedures
- Using more registers for a procedures
 - Solution: stack
 - Restore the old value in a register

Leaf Procedure Example

- C code:

```
int leaf_example (int g, int h, int i,
int j)
{
    int f;
    f = (g + h) - (i + j);
    return f;
}
```

- Arguments g, ..., j in x10, ..., x13
- f in x20
- temporaries x5, x6
- Need to save x5, x6, x20 on stack

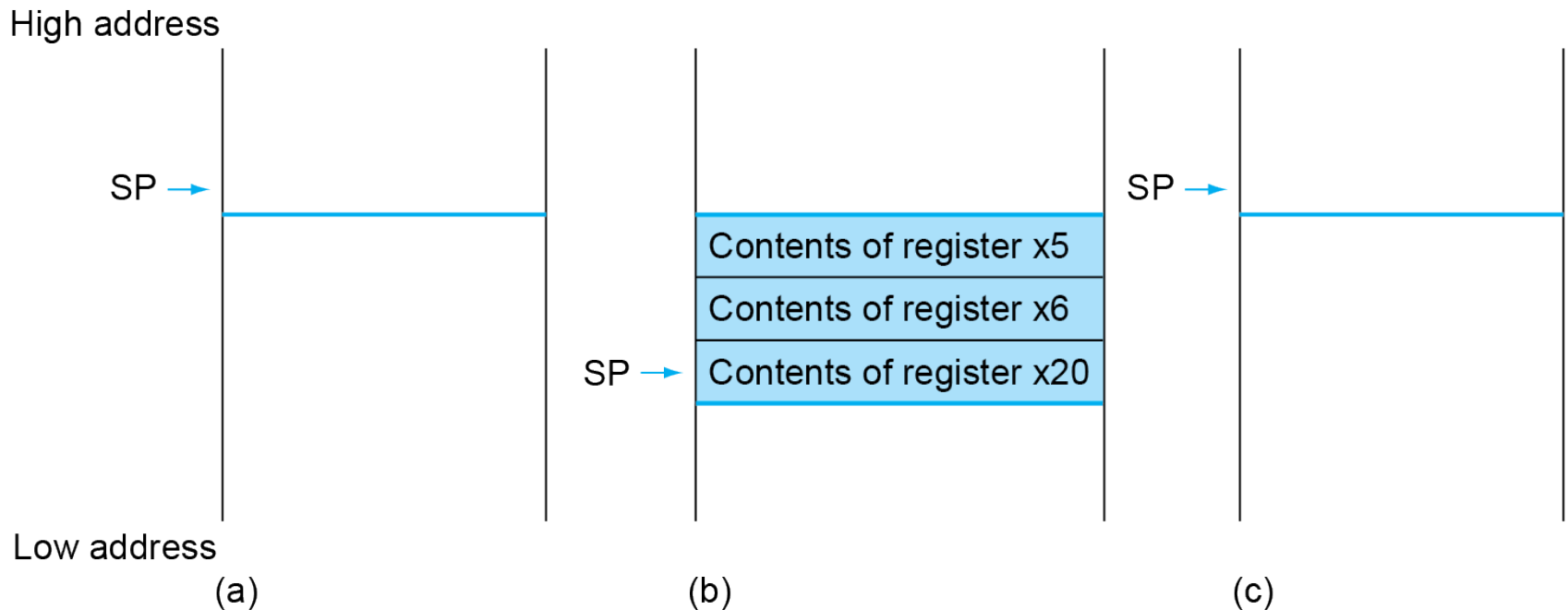
Leaf Procedure Example

■ RISC-V code:

leaf_example:

```
    addi sp, sp, -24    //adjust stack to make room for 3 items
    sw    x5, 16(sp)
    sw    x6, 8(sp)
    sw    x20, 0(sp)    } Save x5, x6, x20 on stack
    add    x5, x10, x11    // x5 = g + h
    add    x6, x12, x1    // x6 = i + j
    sub    x20, x5, x6    // f = x5 - x6
    addi    x10, x20, 0    // copy f to parameter register
    lw    x20, 0(sp)
    lw    x6, 8(sp)
    lw    x5, 16(sp)    } // Restore x5, x6, x20 from stack
    addi    sp, sp, 12    // adjust stack to delete 3 items
    jalr    x0, 0(x1)    // branch back to calling routine
```

Local Data on the Stack



The stack (a) before, (b) during, and (c) after the procedure call

sp points to the top of the stack or the last word in the stack

Register Usage

- x5 – x7, x28 – x31: temporary registers
 - Not preserved by the **callee**
- x8 – x9, x18 – x27: saved registers
 - If used, the **callee** saves and restores them by stack

**Callee: a procedure that executes a series of stored instructions*

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
 - Its return address
 - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
long long int fact (long long int n)
{
    if (n < 1) return f;
    else return n * fact(n - 1);
}
```

- Argument n in x10
- Result in x10

Leaf Procedure Example

■ RISC-V code:

fact:

```
addi sp, sp, -8  
sw   x1, 4(sp)  
sw   x10, 0(sp)
```

Save return address and n on stack

```
addi x5, x10, -1
```

$x5 = n - 1$

```
bge x5, x0, L1
```

if $n \geq 1$, go to L1

```
addi x10, x0, 1
```

Else, set return value to 1

```
addi sp, sp, 8
```

Pop stack, don't bother restoring values

```
jalr x0, 0(x1)
```

Return to caller

```
L1: addi x10, x10, -1
```

$n = n - 1$

```
jal x1, fact
```

call fact($n-1$)

```
addi x6, x10, 0
```

move result of fact($n - 1$) to x6

```
lw x10, 0(sp)
```

Restore caller's n

```
lw x1, 4(sp)
```

Restore caller's return address

```
addi sp, sp, 8
```

Pop stack

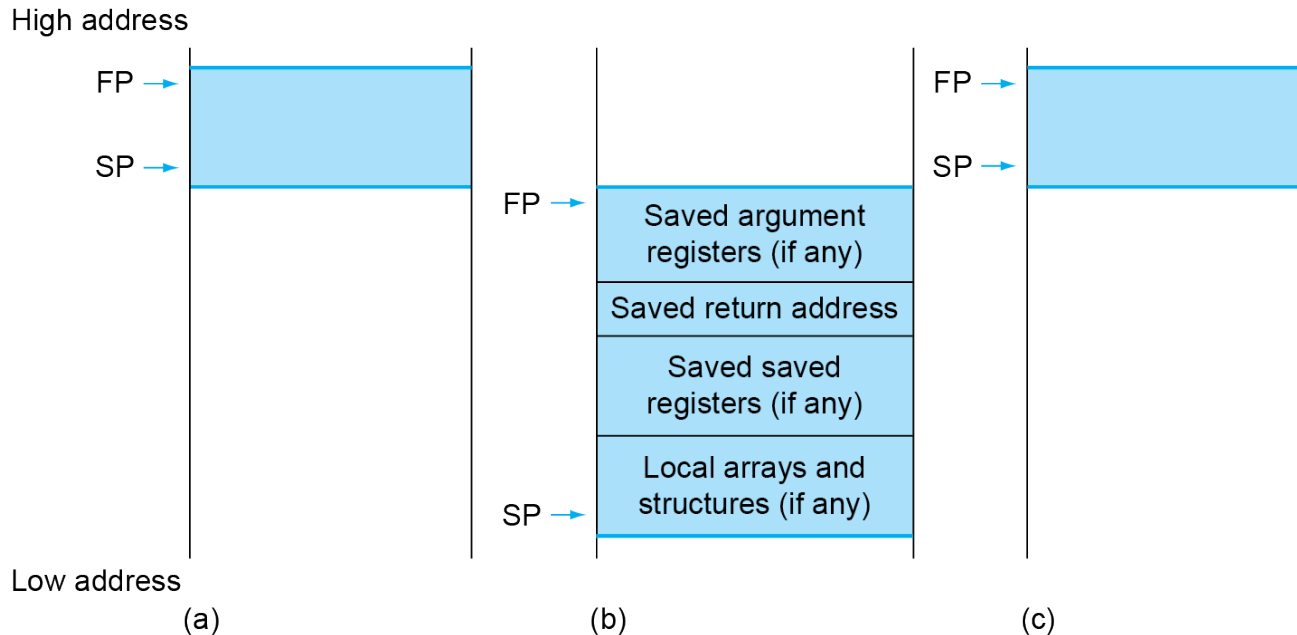
```
mul x10, x10, x6
```

return $n * \text{fact}(n-1)$

```
jalr x0, 0(x1)
```

Return to the caller

Local Data on the Stack



Frame pointer (fp or x8) points to the first word of the frame

- Local data allocated by callee
 - e.g., C automatic variables
- Procedure frame (activation record)
 - Used by some compilers to manage stack storage

Memory Layout

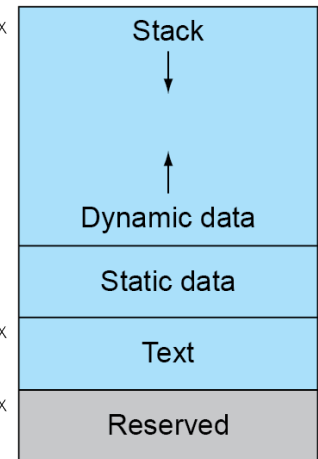
- Text segment: program code
- Static data: global variables
 - e.g., static variables in C, constant arrays and strings
 - x3 (global pointer) initialized to address allowing \pm offsets into this segment
- Dynamic data: heap
 - E.g., malloc in C, new in Java

SP → 0000 003f ffff fff0_{hex}

0000 0000 1000 0000_{hex}

PC → 0000 0000 0040 0000_{hex}

0



Stack: automatic storage

Character Data

- Byte-encoded character sets
 - ASCII: 128 characters
 - 95 graphic, 33 control
 - Latin-1: 256 characters
 - referred to as extended ASCII
- Unicode: 32-bit character set
 - Used in Java, C++ wide characters, ...
 - Most of the world's alphabets, plus symbols
 - UTF-8, UTF-16: variable-length encodings

Character Codes (1)

 **ASCII** stands for **American Standard Code**

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040	 	Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	\$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	((72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051))	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[[123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174	|	
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135]]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177		DEL

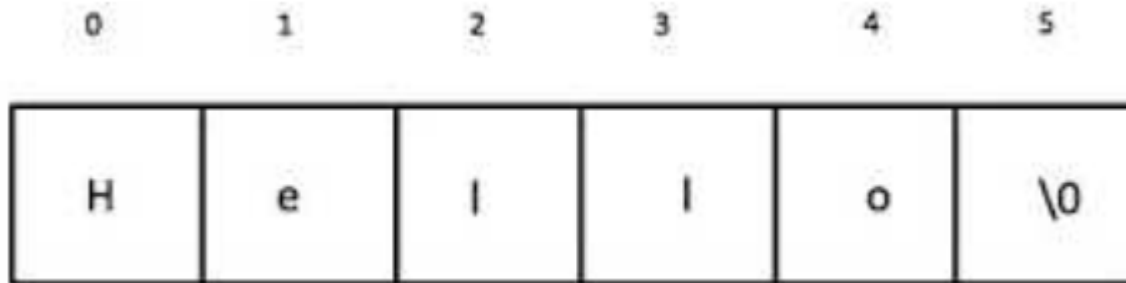
Source: www.LookupTables.com

Character Codes (2)

- The UTF-8 encoding scheme. 1,112,064 valid code
 - The first 128 characters ([US-ASCII](#)) need one byte
 - The next 1,920 characters need two bytes to encode, which covers the remainder of almost all [Latin-script alphabets](#)
 - Three bytes are needed for characters in the rest of the [Basic Multilingual Plane](#) which includes most Chinese, Japanese and Korean characters.
 - Four bytes are needed for characters in the [other planes of Unicode](#) which includes historic scripts, mathematical symbols, and [emoji](#) (pictographic symbols)

Number of bytes	Bits for code point	First code point	Last code point	Byte 1	Byte 2	Byte 3	Byte 4
1	7	U+0000	U+007F	0xxxxxxx			
2	11	U+0080	U+07FF	110xxxxx	10xxxxxx		
3	16	U+0800	U+FFFF	1110xxxx	10xxxxxx	10xxxxxx	
4	21	U+10000	U+10FFFF	11110xxx	10xxxxxx	10xxxxxx	10xxxxxx

Representing Strings



② Strings are represented by a sequence of characters.

② Each character is encoded in **ASCII format**.

② Strings should be **null-terminated**. That is, the final character has ASCII code 0. I.e., a string of k chars requires $k + 1$ bytes.

Example of "Hello!" in C

```
jsun@jsun-ThinkPad-T460p:~/cmpe120$ cat ch1_ascii_string.c
#include <stdio.h>
#include <string.h>

char * str = "Hello!";

void main() {
    int i;
    printf("%s\n", str);

    for (i=0; i<strlen(str)+1; i++)
        printf("%c(%02x, %d) ", str[i], str[i], str[i]);
    printf("\n");
}
jsun@jsun-ThinkPad-T460p:~/cmpe120$ gcc ch1_ascii_string.c
jsun@jsun-ThinkPad-T460p:~/cmpe120$ ./a.out
Hello!
H(48, 72) e(65, 101) l(6c, 108) l(6c, 108) o(6f, 111) !(21, 33) (00, 0)
```

H	e	l	l	o	!	\0
48	65	6c	6c	6f	21	0

Byte/Halfword/Word Operations

- RISC-V byte/halfword/word load/store
 - Load byte/halfword/word: Sign extend to 32 bits in rd
 - `lb rd, offset(rs1)`
 - `lh rd, offset(rs1)`
 - `lw rd, offset(rs1)`
 - Load byte/halfword/word unsigned: Zero extend to 32 bits in rd
 - `lbu rd, offset(rs1)`
 - `lhu rd, offset(rs1)`
 - Store byte/halfword/word: Store rightmost 8/16/32 bits
 - `sb rs2, offset(rs1)`
 - `sh rs2, offset(rs1)`
 - `sw rs2, offset(rs1)`

String Copy Example

- C code:

- Null-terminated string

```
void strcpy (char x[], char y[])
{ size_t i;
  i = 0;
  while ((x[i]=y[i])!='\0')
    i += 1;
}
```

Base addresses of array x and y in x10 and x11
i in the x19

String Copy Example

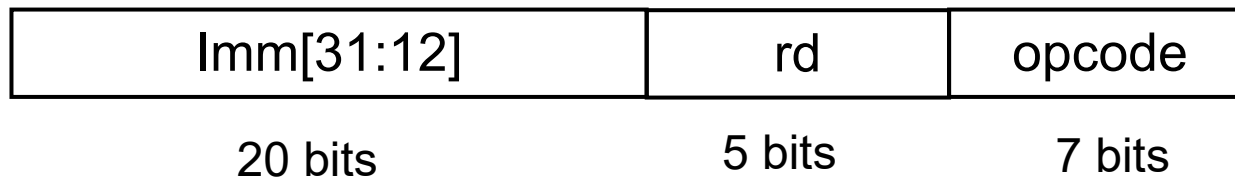
■ RISC-V code:

strcpy:

```
    addi sp,sp,-4           // adjust stack for 1 word
    sd   x19,0(sp)         // push x19
    add  x19,x0,x0         // i=0
L1:  add  x5,x19,x11        // x5 = addr of y[i]
     lbu  x6,0(x5)         // x6 = y[i]
     add  x7,x19,x10       // x7 = addr of x[i]
     sb   x6,0(x7)         // x[i] = y[i]
     beq  x6,x0,L2         // if y[i] == 0 then exit
     addi x19,x19,1        // i = i + 1
     jal  x0,L1            // next iteration of loop
L2:  ld   x19,0(sp)        // restore saved x19
     addi sp,sp,4          // pop 1 doubleword from stack
     jalr x0,0(x1)         // and return
```

32-bit Constants

- Most constants are small
 - 12-bit immediate is sufficient
- For the occasional 32-bit constant
 - `lui rd, constant`
 - Copies 20-bit constant to bits [31:12] of `rd`
 - Fill bits [11:0] of `rd` with 0
- U-type format



Example: Loading a 32-bit Constant

- What is the RISC-V assembly code to load this 32-bit constant into register x19?

0000 0000 0011 1101 0000 0101 0000 0000

Load bits 12 through 31

```
lui x19, 976 // 0000 0000 0011 1101 0000
```

0000 0000 0011 1101 0000	0000 0000 0000
--------------------------	----------------

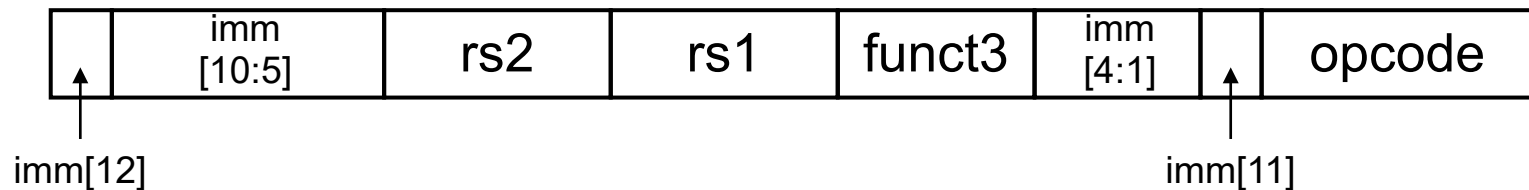
Add in the lowest 12 bits

```
addi x19,x19,1280 // 0000 0101 0000 0000
```

0000 0000 0011 1101 0000	0101 0000 0000
--------------------------	----------------

Branch Addressing

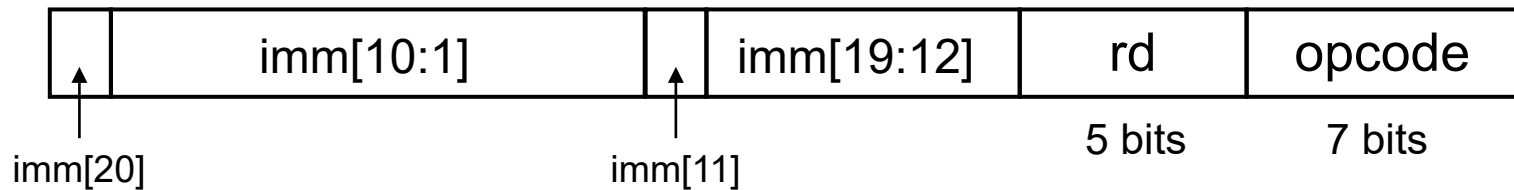
- Branch instructions specify
 - Opcode, two registers, target address
- Most branch targets are near branch
 - Forward or backward
- SB format:



- PC-relative addressing
 - Target address = PC + branch offset

Jump Addressing

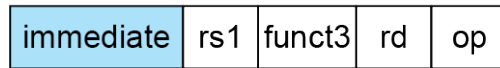
- Jump and link (jal) target uses 20-bit immediate for larger range
- UJ format:



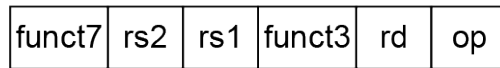
- For long jumps, eg, to 32-bit absolute address
 - lui: load address[31:12] to temp register
 - jalr: add address[11:0] and jump to target

RISC-V Addressing Summary

1. Immediate addressing



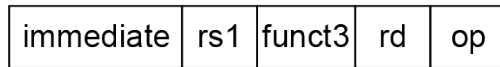
2. Register addressing



Registers

Register

3. Base addressing



Memory

Register

+

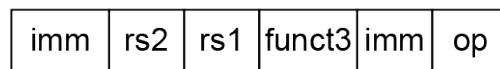
Byte

Halfword

Word

Doubleword

4. PC-relative addressing



Memory

PC

+

Word

RISC-V Encoding Summary

Name (Field Size)	Field					Comments	
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Synchronization

- Two processors sharing an area of memory
 - P1 writes, then P2 reads
 - Data race if P1 and P2 don't synchronize
 - Result depends of order of accesses
- Hardware support required
 - Atomic read/write memory operation
 - No other access to the location allowed between the read and write
- Could be a single instruction
 - E.g., atomic swap of register \leftrightarrow memory
 - Or an atomic pair of instructions

Synchronization in RISC-V

- Load reserved: `lr.w rd, (rs1)`
 - Load from address in rs1 to rd
 - Place reservation on memory address
- Store conditional: `sc.w rd, (rs1), rs2`
 - Store from rs2 to address in rs1
 - Succeeds if location not changed since the `lr.w`
 - Returns 0 in rd
 - Fails if location is changed
 - Returns non-zero value in rd

Synchronization in RISC-V

- Example 1: atomic swap (to test/set lock variable)

```
again: lr.w x10,(x20)
       sc.w x11,(x20),x23 // x11 = status
       bne x11,x0,again   // branch if store failed
       addi x23,x10,0     // x23 = loaded value
```

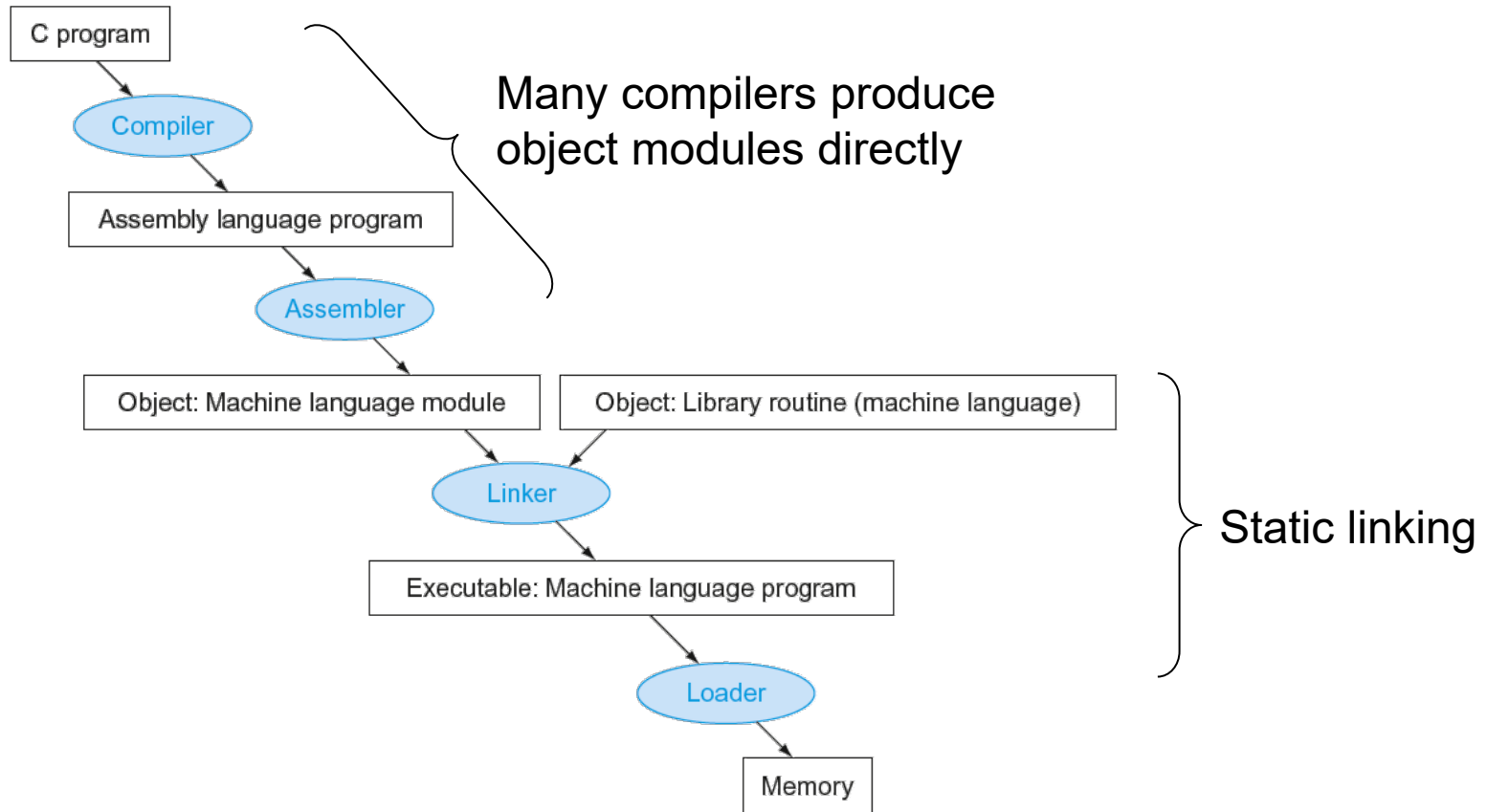
- Example 2: lock

```
       addi x12,x0,1      // copy locked value
again: lr.d x10,(x20)     // read lock
       bne x10,x0,again   // check if it is 0 yet
       sc.d x11,(x20),x12 // attempt to store
       bne x11,x0,again   // branch if fails
```

- Unlock:

```
sd     x0,0(x20)         // free lock
```

Translation and Startup



simple-op-main.c

- Get two unsigned integers from user
- Call simple_op with these 2 parameters
- Print out results

```
debian@debian-mipsel: ~/ch2-examples/simple-op
#include <stdio.h>

extern unsigned int simple_op(unsigned int a, unsigned int b);

void main() {
    int a, b, result;

    printf("Please input a : ");
    scanf("%u", &a);

    printf("Please input b : ");
    scanf("%u", &b);

    result = simple_op(a, b);
    printf("simple_op(%u, %u) = %u(0x%08x)\n", a, b, result, result);
}
```

simple-op.S

- Capital suffix of "S" is by convention for human written assembly code
 - It also go through preprocessor for external file inclusion etc.

directives

comments

```
debian@debian-mipsel: ~/ch2-examples/simple-op
# A simple function implemented in assembly, called simple_op

.text
.global simple_op
.type    simple_op, @function
simple_op:
add      $v0,$a0,$a1    # add 2 args and return sum
jr       $ra            # return to caller
```

labels

code

Jump
register
instruction

Producing an Object Module

- Assembler (or compiler) translates program into machine instructions
- Provides information for building a complete program from the pieces
 - Header: described contents of object module
 - Text segment: translated instructions
 - Static data segment: data allocated for the life of the program
 - Relocation info: for contents that depend on absolute location of loaded program
 - Symbol table: global definitions and external refs
 - Debug info: for associating with source code

Linking Object Modules

- Produces an executable file
 1. Merges segments
 2. Resolve labels (determine their addresses)
 3. Patch location-dependent and external refs
- Could leave location dependencies for fixing by a relocating loader
 - But with virtual memory, no need to do this
 - Program can be loaded into absolute location in virtual memory space

Loading a Program

- Load from image file on disk into memory
 1. Read header to determine segment sizes
 2. Create virtual address space
 3. Copy text and initialized data into memory
 - Or set page table entries so they can be faulted in
 4. Set up arguments on stack
 5. Initialize registers (including sp, fp, gp)
 6. Jump to startup routine
 - Copies arguments to x10, ... and calls main
 - When main returns, do exit syscall

Dynamic Linking

- Problem on traditional approach- linking libraries before the program is run
 - If new version of the library is released
 - Load all routines in the library, which can be large vs. the programs
- Solution - Dynamic Linking
 - Only link/load library procedure when it is called
 - Requires procedure code to be relocatable
 - Avoids image bloat caused by static linking of all (transitively) referenced libraries
 - Automatically picks up new library versions

C Sort Example

- Illustrates use of assembly instructions for a C bubble sort function
- Swap procedure (leaf)

```
void swap(int v[],
          size_t int k)
{
    int temp;
    temp = v[k];
    v[k] = v[k+1];
    v[k+1] = temp;
}
```

- v in x10, k in x11, temp in x5

The Procedure Swap

swap:

```
slli x6,x11,2    // reg x6 = k * 4
add  x6,x10,x6    // reg x6 = v + (k * 4)
lw   x5,0(x6)     // reg x5 (temp) = v[k]
lw   x7,4(x6)     // reg x7 = v[k + 1]
sd   x7,0(x6)     // v[k] = reg x7
sd   x5,4(x6)     // v[k+1] = reg x5 (temp)
jalr x0,0(x1)     // return to calling routine
```


The Sort Procedure in C

- Non-leaf (calls swap)

```
void sort (int v[], size_t n)
{
    size_t i, j;
    for (i = 0; i < n; i += 1) {
        for (j = i - 1;
             j >= 0 && v[j] > v[j + 1];
             j -= 1) {
            swap(v, j);
        }
    }
}
```

- v in x10, n in x11, i in x19, j in x20

The Outer Loop

- Skeleton of outer loop:

- for (i = 0; i < n; i += 1) {

```
addi    x19,x0, 0           // i = 0
```

```
for1tst:
```

```
bge    x19,x11,exit1    // go to exit1 if x19 ≥ x11 (i≥n)
```

...

(body of outer for-loop)

```
addi x19,x19,1      // i += 1
```

```
jal x0, for1tst      // branch to test of outer loop
```

```
exit1:
```

The Inner Loop

- Skeleton of inner loop:

- `for (j = i - 1; j >= 0 && v[j] > v[j + 1]; j -= 1) {`

```
    addi x20,x19,-1    // j = i - 1
```

```
for2tst:
```

```
    blt  x20,x0,exit2  // go to exit2 if x20 < 0 (j < 0)
```

```
    slli x5,x20,2      // reg x5 = j * 4
```

```
    add  x5,x10,x5     // reg x5 = v + (j * 4)
```

```
    lw   x6,0(x5)      // reg x6 = v[j]
```

```
    lw   x7,4(x5)      // reg x7 = v[j + 1]
```

```
    ble  x6,x7,exit2   // go to exit2 if x6 ≤ x7
```

```
    ...
```

```
    (body of second for loop)
```

```
    ...
```

```
    addi x20,x20,-1    // j -= 1
```

```
    jal  x0, for2tst    // branch to test of inner loop
```

```
exit2:
```

Preserving Registers

■ Preserve saved registers:

```
addi sp,sp,-20 // make room on stack for 5 regs
sw   x1,16(sp) // save x1 on stack
sw   x22,12(sp) // save x22 on stack
sw   x21,8(sp)  // save x21 on stack
sw   x20,4(sp)  // save x20 on stack
sw   x19,0(sp)  // save x19 on stack
```

■ Restore saved registers:

exit1:

```
sw   x19,0(sp) // restore x19 from stack
sw   x20,4(sp) // restore x20 from stack
sw   x21,8(sp) // restore x21 from stack
sw   x22,12(sp) // restore x22 from stack
sw   x1,16(sp) // restore x1 from stack
addi sp,sp, 20 // restore stack pointer
jalr x0,0(x1)
```

Arrays vs. Pointers

- Array indexing involves
 - Multiplying index by element size
 - Adding to array base address
- Pointers correspond directly to memory addresses
 - Can avoid indexing complexity

Example: Clearing an Array

```
clear1(int array[], int size) {  
    int i;  
    for (i = 0; i < size; i += 1)  
        array[i] = 0;  
}
```

```
    addi x5, x0, 0          // i = 0  
loop1:  
    slli x6, x5, 2          // x6 = i * 4  
    add x7, x10, x6         // x7 = address  
                        // of array[i]  
    sw x0, 0(x7)           // array[i] = 0  
    addi x5, x5, 1          // i = i + 1  
    blt x5, x11, loop1      // if (i < size)  
                        // go to loop1
```

```
clear2(int *array, int size) {  
    int *p;  
    for (p = &array[0]; p < &array[size];  
        p = p + 1)  
        *p = 0;  
}
```

```
    add x5, x10, x0         // p = address  
                        // of array[0]  
    slli x6, x11, 2         // x6 = size * 2  
    add x7, x10, x6         // x7 = address  
                        // of array[size]  
loop2:  
    sw x0, 0(x5)           // Memory[p] = 0  
    addi x5, x5, 4          // p = p + 4  
    bltu x5, x7, loop2      // if (p < &array[size])  
                        // go to loop2
```

Comparison of Array vs. Ptr

- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
 - Part of index calculation for incremented i
 - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
 - Induction variable elimination
 - Better to make program clearer and safer

MIPS Instructions

- MIPS: commercial predecessor to RISC-V
- Similar basic set of instructions
 - 32-bit instructions
 - 32 general purpose registers, register 0 is always 0
 - 32 floating-point registers
 - Memory accessed only by load/store instructions
 - Consistent use of addressing modes for all data sizes
- Different conditional branches
 - For $<$, $<=$, $>$, $>=$
 - RISC-V: `blt`, `bge`, `bltu`, `bgeu`
 - MIPS: `slt`, `sltu` (set less than, result is 0 or 1)
 - Then use `beq`, `bne` to complete the branch

Instruction Encoding

Register-register

	31	25	24	20	19	15	14	12	11	7	6	0																		
RISC-V	funct7(7)					rs2(5)				rs1(5)				funct3(3)		rd(5)		opcode(7)												
	31	26	25	21	20	16	15	11	10	6	5	0																		
MIPS	Op(6)					Rs1(5)					Rs2(5)					Rd(5)					Const(5)					Opx(6)				

Load

	31	20	19	15	14	12	11	7	6	0										
RISC-V	immediate(12)					rs1(5)			rd(5)		opcode(7)									
	31	26	25	21	20	16	15					0								
MIPS	Op(6)					Rs1(5)					Rs2(5)					Const(16)				

Store

	31	25	24	20	19	15	14	12	11	7	6	0											
RISC-V	immediate(7)					rs2(5)				rs1(5)			funct3(3)		immediate(5)		opcode(7)						
	31	26	25	21	20	16	15											0					
MIPS	Op(6)					Rs1(5)					Rs2(5)					Const(16)							

Branch

	31	25	24	20	19	15	14	12	11	7	6	0								
RISC-V	immediate(7)					rs2(5)			rs1(5)			funct3(3)		immediate(5)		opcode(7)				
	31	26	25	21	20	16	15													
MIPS	Op(6)					Rs1(5)					Opx/Rs2(5)					Const(16)				

The Intel x86 ISA

- Evolution with backward compatibility
 - 8080 (1974): 8-bit microprocessor
 - Accumulator, plus 3 index-register pairs
 - 8086 (1978): 16-bit extension to 8080
 - Complex instruction set (CISC)
 - 8087 (1980): floating-point coprocessor
 - Adds FP instructions and register stack
 - 80286 (1982): 24-bit addresses, MMU
 - Segmented memory mapping and protection
 - 80386 (1985): 32-bit extension (now IA-32)
 - Additional addressing modes and operations
 - Paged memory mapping as well as segments

The Intel x86 ISA

- Further evolution...
 - i486 (1989): pipelined, on-chip caches and FPU
 - Compatible competitors: AMD, Cyrix, ...
 - Pentium (1993): superscalar, 64-bit datapath
 - Later versions added MMX (Multi-Media eXtension) instructions
 - The infamous FDIV bug
 - Pentium Pro (1995), Pentium II (1997)
 - New microarchitecture (see Colwell, *The Pentium Chronicles*)
 - Pentium III (1999)
 - Added SSE (Streaming SIMD Extensions) and associated registers
 - Pentium 4 (2001)
 - New microarchitecture
 - Added SSE2 instructions

The Intel x86 ISA

- And further...
 - AMD64 (2003): extended architecture to 64 bits
 - EM64T – Extended Memory 64 Technology (2004)
 - AMD64 adopted by Intel (with refinements)
 - Added SSE3 instructions
 - Intel Core (2006)
 - Added SSE4 instructions, virtual machine support
 - AMD64 (announced 2007): SSE5 instructions
 - Intel declined to follow, instead...
 - Advanced Vector Extension (announced 2008)
 - Longer SSE registers, more instructions
- If Intel didn't extend with compatibility, its competitors would!
 - Technical elegance ≠ market success

Basic x86 Registers

Name	31	0	Use
EAX			GPR 0
ECX			GPR 1
EDX			GPR 2
EBX			GPR 3
ESP			GPR 4
EBP			GPR 5
ESI			GPR 6
EDI			GPR 7
	CS		Code segment pointer
	SS		Stack segment pointer (top of stack)
	DS		Data segment pointer 0
	ES		Data segment pointer 1
	FS		Data segment pointer 2
	GS		Data segment pointer 3
EIP			Instruction pointer (PC)
EFLAGS			Condition codes

Basic x86 Addressing Modes

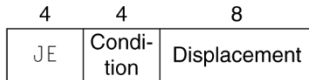
- Two operands per instruction

Source/dest operand	Second source operand
Register	Register
Register	Immediate
Register	Memory
Memory	Register
Memory	Immediate

- Memory addressing modes
 - Address in register
 - $\text{Address} = R_{\text{base}} + \text{displacement}$
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}}$ (scale = 0, 1, 2, or 3)
 - $\text{Address} = R_{\text{base}} + 2^{\text{scale}} \times R_{\text{index}} + \text{displacement}$

x86 Instruction Encoding

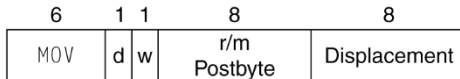
a. JE EIP + displacement



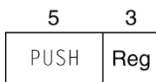
b. CALL



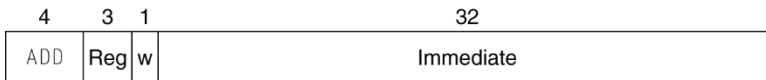
c. MOV EBX, [EDI + 45]



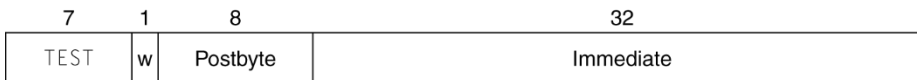
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



■ Variable length encoding

- Postfix bytes specify addressing mode
- Prefix bytes modify operation
 - Operand length, repetition, locking, ...

Implementing IA-32

- Complex instruction set makes implementation difficult
 - Hardware translates instructions to simpler microoperations
 - Simple instructions: 1–1
 - Complex instructions: 1–many
 - Microengine similar to RISC
 - Market share makes this economically viable
- Comparable performance to RISC
 - Compilers avoid complex instructions

Other RISC-V Instructions

- Base integer instructions (RV64I)
 - Those previously described, plus
 - `auipc rd, imm` // $rd = (imm \ll 12) + pc$
 - follow by `jalr` (adds 12-bit `imm`) for long jump
 - `slt`, `sltu`, `slti`, `sltui`: set less than (like MIPS)
- 32-bit variant: RV32I
 - registers are 32-bits wide, 32-bit operations

Instruction Set Extensions

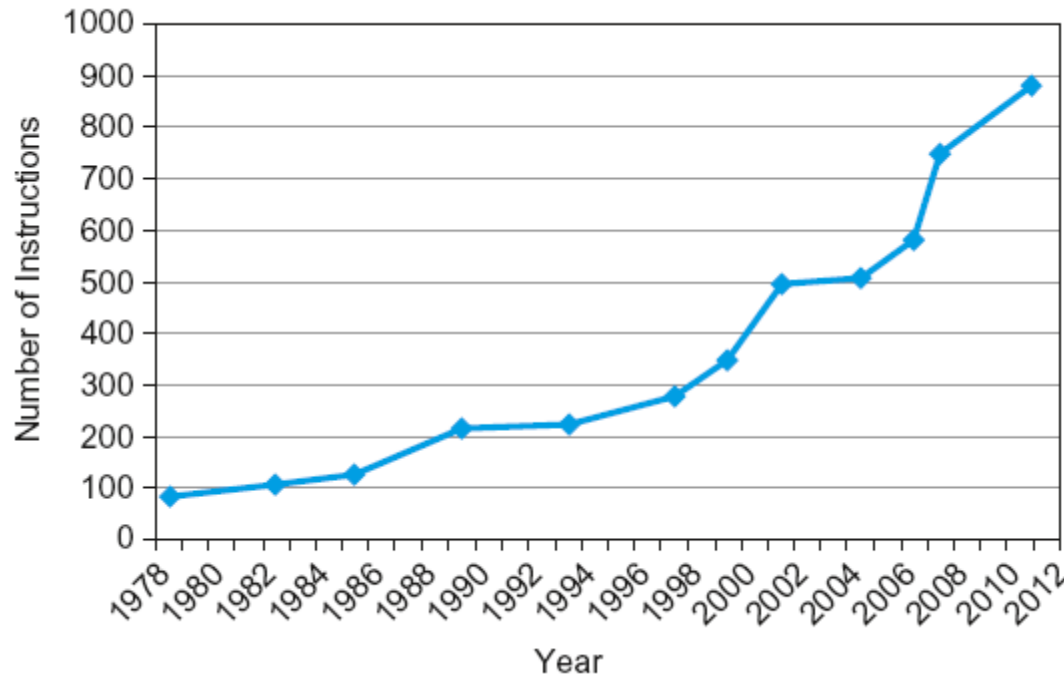
- M: integer multiply, divide, remainder
- A: atomic memory operations
- F: single-precision floating point
- D: double-precision floating point
- C: compressed instructions
 - 16-bit encoding for frequently used instructions

Fallacies

- Powerful instruction \Rightarrow higher performance
 - Fewer instructions required
 - But complex instructions are hard to implement
 - May slow down all instructions, including simple ones
 - Compilers are good at making fast code from simple instructions
- Use assembly code for high performance
 - But modern compilers are better at dealing with modern processors
 - More lines of code \Rightarrow more errors and less productivity

Fallacies

- Backward compatibility \Rightarrow instruction set doesn't change
 - But they do accrete more instructions



x86 instruction set

Pitfalls

- Sequential words are not at sequential addresses
 - Increment by 4, not by 1!
- Keeping a pointer to an automatic variable after procedure returns
 - e.g., passing pointer back via an argument
 - Pointer becomes invalid when stack popped

Concluding Remarks

- Design principles
 1. Simplicity favors regularity
 - Keep all instruction in a single size
 2. Smaller is faster
 - only 32 registers
 3. Good design demands good compromises
 - Larger the addresses and constants and keep all in same size

Concluding Remarks

- Make the common case fast
 - PC-relative addressing
- Abstraction
 - Layers of software/hardware
 - Compiler, assembler, hardware