CS202 : COMPUTER ORGANIZATION

Lecture 2
Instruction Set Architecture (1)

2023 Spring

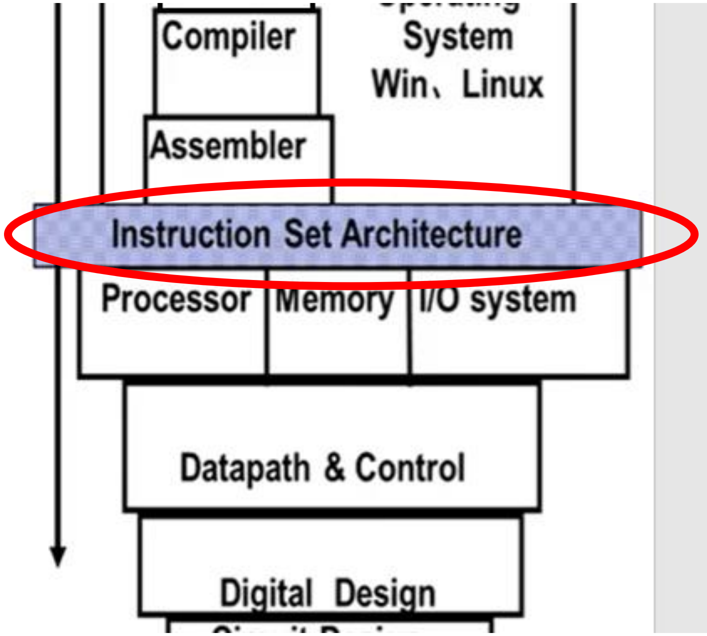
Today’s Agenda

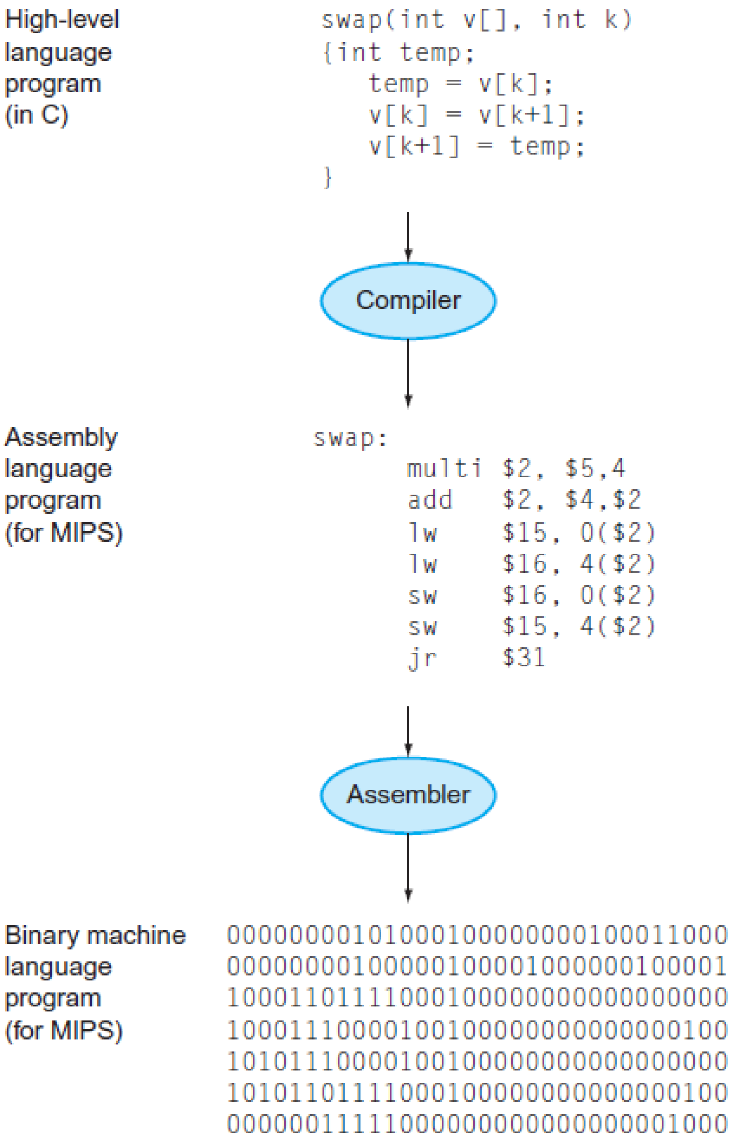
- Recap
- Context
  - Instruction set architecture
  - Operands
    - Register operands and their organization
    - Memory operands, data transfer
    - Immediate operands
  - Signed and unsigned numbers
  - Representing instructions
  - Operations
    - Logical
    - Conditional
- Reading: Textbook, Sections 2.1 - 2.7

Instruction Set Architecture (ISA)

- Computer Architecture
  - Instruction Set Architecture
  - Machine Organization

- ISA => Assembly Language
  - From programmer’s point of view




High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

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

```
swap:
      multi $2, $5,4
      add   $2, $4,$2
      lw    $15, 0($2)
      lw    $16, 4($2)
      sw    $16, 0($2)
      sw    $15, 4($2)
      jr    $31
```

```
00000000101000100000000100011000
00000000100000100001000000100001
10001101111100010000000000000000
10001110000100100000000000000100
10101110000100100000000000000000
10101101111100010000000000000100
00000011111000000000000000001000
```

Recall in High Level Language

- Operators: +, -, *, /, % (mod), ...
  - 7/4==1, 7%4==3
- Operands:
  - Variables: lower, upper, fahr, celsius
  - Constants: 0, 1000, -17, 15.4
- Assignment statement:

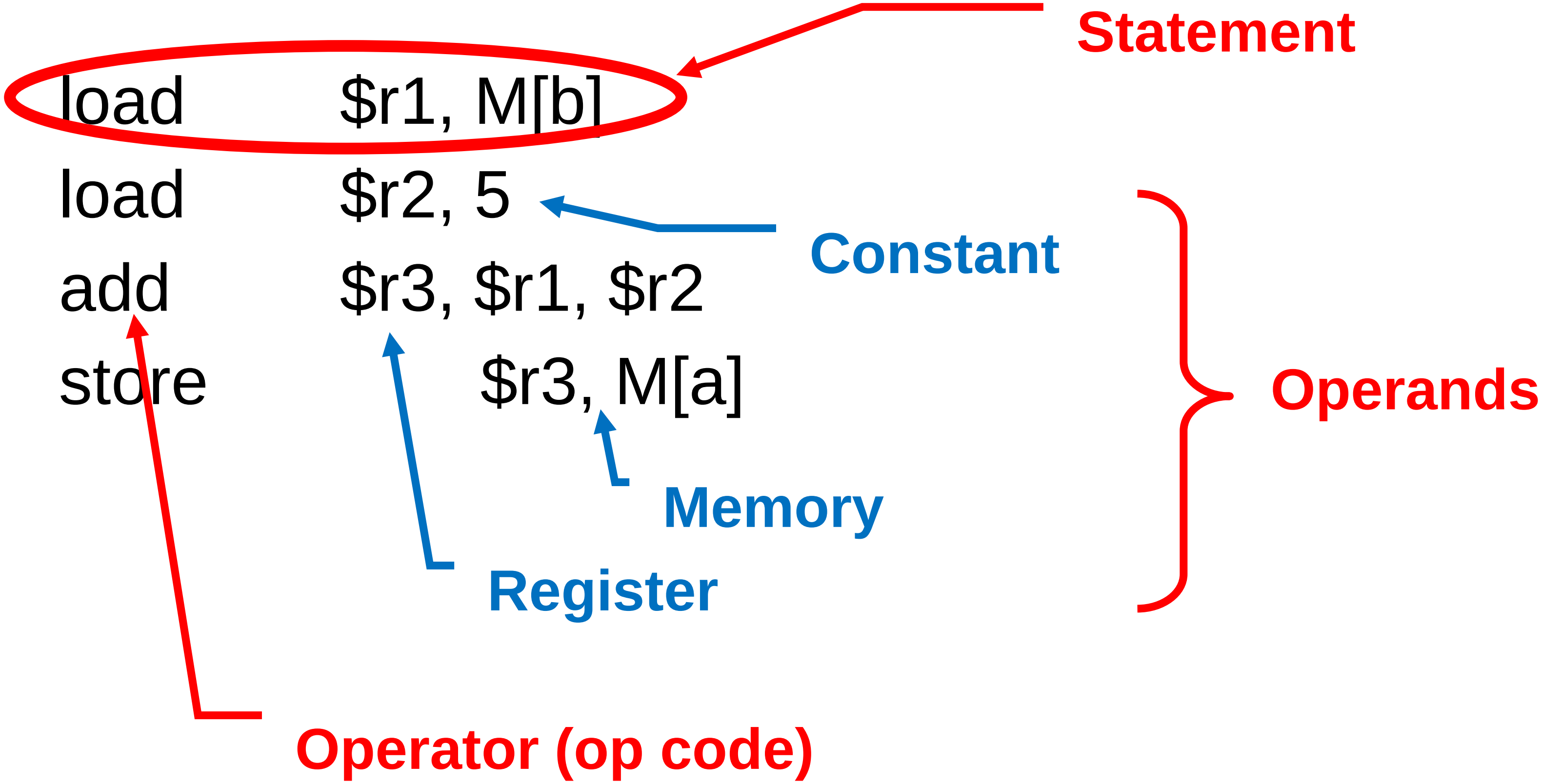
  variable = expression

  - Expressions consist of operators operating on operands, e.g.,

```
celsius = 5*(fahr-32)/9;
a = b+c+d-e;
```

When Translating to Assembly

a = b + 5;



```
load     $r1, M[b]
load     $r2, 5
add      $r3, $r1, $r2
store        $r3, M[a]
```

Statement

Constant

Operands

Memory

Register

Operator (op code)

Components of an ISA

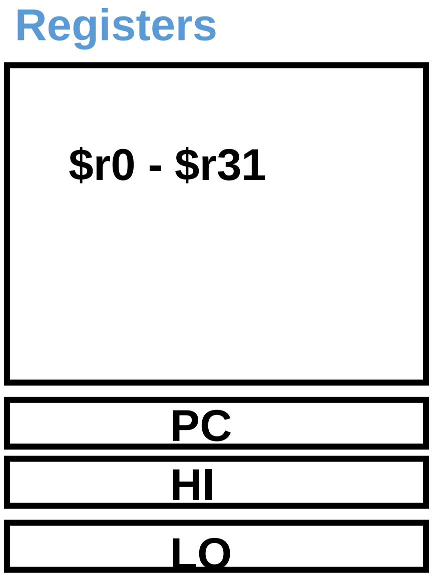
- Organization of programmable storage
  - registers
  - memory
  - modes of addressing and accessing data items and instructions
- Data types and data structures
  - encoding and representation
- Instruction formats
- Instruction set (or operation code)
  - ALU, control transfer, exceptional handling
- Different computers have different instruction sets
  - But with many aspects in common
  - Early computers had very simple instruction sets
    - Simplified implementation
  - Many modern computers also have simple instruction sets

RISC vs. CISC ISA

- RISC
  - Reduced Instruction Set Computer
  - Fixed instruction size
  - Simple instructions (load/store)
  - Emphasizes more on software (compiler)
  - e.g. MIPS, ARM, PowerPC, RISC-V
- CISC
  - Complex Instruction Set Computer
  - Variable instruction length
  - Much more powerful instructions
  - Hardware intensive instructions (more transistors
  - e.g. x86

The MIPS Instruction Set

- All instruction set are similar
  - Will use MIPS throughout the book
- MIPS
  - Stanford, commercialized by MIPS Technologies
  - Large share of embedded core market
  - Applications in consumer electronics,…
- Instruction categories:
  - Load/Store
  - Computation
  - Jump and Branch
  - Floating Point
  - etc.



Registers

$r0 - $r31

PC

HI

LO

OP	$rs	$rt	$rd	shamt	funct

OP	$rs	$rt	immediate

OP	jump target

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 favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
- Example
  - C code: f = (g + h) - (i + j);
  - Compiled MIPS code:

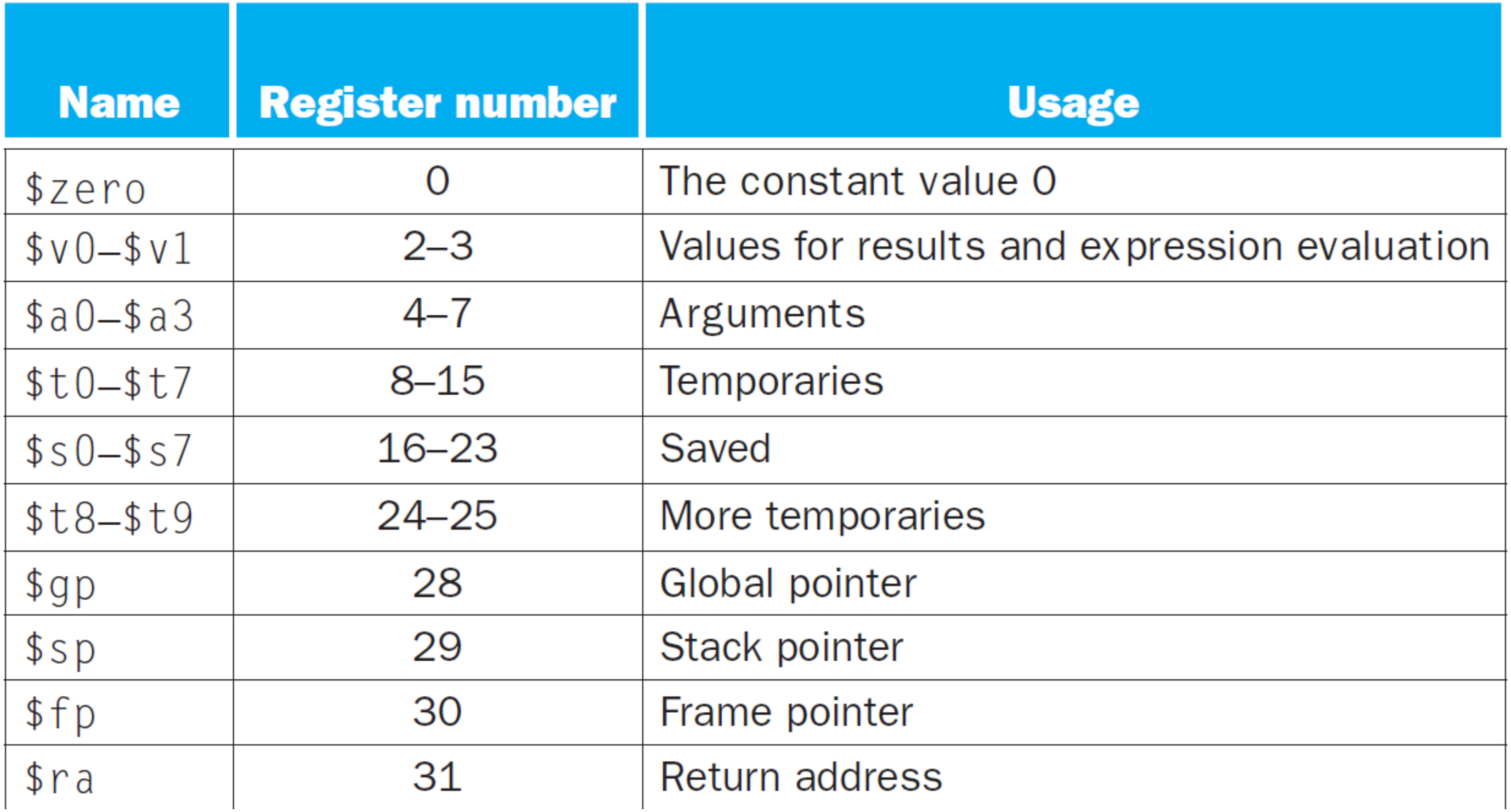
```
add  t0, g,  h      # temp t0 = g + h
add  t1, i,  j      # temp t1 = i + j
sub  f,  t0, t1     # f = t0 - t1
```

Register Operands

- Arithmetic instructions use register operands
- MIPS has a 32 × 32-bit register file
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- Assembler names
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations

An overview of MIPS registers

- 32 × 32-bit registers



Name	Register number	Usage
$zero	0	The constant value 0
$v0–$v1	2–3	Values for results and expression evaluation
$a0–$a3	4–7	Arguments
$t0–$t7	8–15	Temporaries
$s0–$s7	16–23	Saved
$t8–$t9	24–25	More temporaries
$gp	28	Global pointer
$sp	29	Stack pointer
$fp	30	Frame pointer
$ra	31	Return address

Arithmetic Instructions

Category	Instruction	Example	Meaning
Arithmetic	add	add rd, rs, rt	rd ← rs + rt
	subtract	sub rd, rs, rt	rd ← rs – rt

- C code: f = (g + h) - (i + j);
  - Suppose f, …, j in $s0, …, $s4
- Compiled MIPS code:
  - Step 1: Specify registers containing variables
  - Step 2: Express instruction in MIPS

```
add $t0,$s1,$s2
add $t1,$s3,$s4
sub $s0,$t0,$t1
```

$s0	$s1	$s2	$s3	$s4	$s5	$s6	$s7
f	g	h	i	j			

$t0	$t1	$t2	$t3	$t4	$t5	$t6	$t7	$t8	$t9
g+h	i+j								

Immediate Operands

Category	Instruction	Example	Meaning
Arithmetic	add immediate	addi rt, rs, 20	rt = rs + 20

- C code: a++;
  - Suppose a in $s3
- Constant data specified in an instruction

```
    addi $s3, $s3, 1
```

- No subtract immediate instruction
  - Just use a negative constant

```
    addi $s2, $s1, -1
```

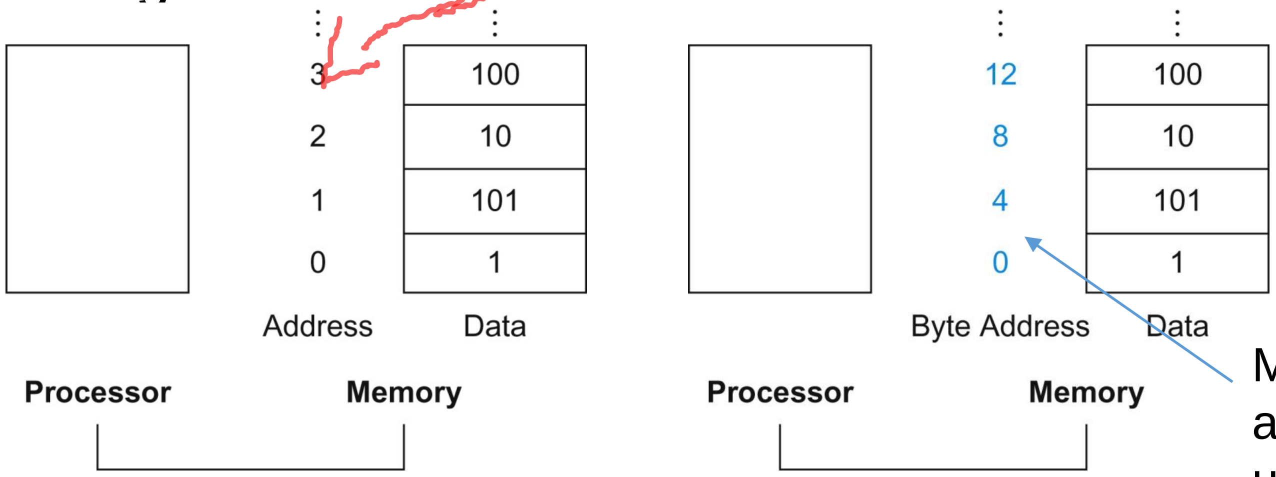
等价

- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Memory Operands

存储

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - ① Load values from memory into registers
  - Store result from register to memory
- In MIPS, each cell is 1 word (4 bytes) long
- Words are aligned in memory
  - Address must be a multiple of 4
- MIPS is Big Endian



Memory address is in unit of byte

Endianness

- Byte order: numbering of bytes within a word
  - Big Endian: most-significant byte at least address of a word
    - IBM 360/370, MIPS
  - Little Endian: least-significant byte at least address
    - x86, RISC-V

Address 4	
Address 3	Byte 0
Address 2	Byte 1
Address 1	Byte 2
Address 0	Byte 3

Big endian
MSB

Address 4	
Address 3	Byte 3
Address 2	Byte 2
Address 1	Byte 1
Address 0	Byte 0

Little endian
MSB

Memory Operand Example

- C code: g = h + A[8];
  - Suppose g in $s1
  - h in $s2,
  - base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
  - 4 bytes per word

```
lw  $t0, 32($s3) #load word
add $s1, $s2, $t0
```

offset

偏移量

base register

…	…	…
A[8]	base+32	100
…	…	…
A[1]	base+4	101
A[0]	base addr	1

Data Transfer Instructions

Category	Instruction	Example	Meaning
Data transfer	load word	lw rt, 20(rs)	rt ← Memory[rs + 20]
	Store word	sw rt, 20(rs)	Memory[rs + 20] ← rt

- C code: A[12] = h + A[8];
  - h in $s2
  - base address of A in $s3
- Compiled MIPS code:
  - Index 8 requires offset of 32
  - What’s the MIPS code?

```
lw  $t0, 32($s3)     # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)     # store word
```

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!

The Constant Zero

- $t2←$s1?
  - *addi $t2, $s1, 0*
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - Move between registers

  add $t2, $s1, $zero
  - Load constant 10 into $s2

  addi $s2, $zero, 10

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1 \times 2^{31} + 1 \times 2^{30} + \ldots + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

Signed Negation

- Complement and add 1 反码+1
  - Complement means 1 → 0, 0 → 1

$$x + \overline{x} = 1111...111_2 = -1$$

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

- Example: negate +2
  - $+2 = 0000\ 0000 \ldots 0010_2$
  - $-2 = 1111\ 1111 \ldots 1101_2 + 1$
    $= 1111\ 1111 \ldots 1110_2$

Sign Extension

符号扩充

- Representing a number using more bits
  - Preserve the numeric value
- In MIPS instruction set
  - `addi`: extend immediate value
  - `lb`, `lh`: extend loaded byte/halfword
  - `beq`, `bne`: extend the displacement
- 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

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- MIPS instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!
- Register numbers
  - $t0 – $t7 are reg’s 8 – 15
  - $t8 – $t9 are reg’s 24 – 25
  - $s0 – $s7 are reg’s 16 – 23

MIPS R-format Instructions

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- Instruction fields
  - op: operation code (opcode)
  - rs: first source register number($s1)
  - rt: second source register number ($s2)
  - rd: destination register number ($t0)
  - shamt: shift amount (00000 for now)
  - funct: function code (extends opcode)

R-format Example

Register numbers
    $t0 – $t7 are reg's 8 – 15
    $t8 – $t9 are reg's 24 – 25
    $s0 – $s7 are reg's 16 – 23

```
add $t0, $s1, $s2
```

op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$$00000010001100100100000000100000_2 = 02324020_{16}$$

MIPS I-format Instructions

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits

- Immediate arithmetic and load/store instructions
  - rt: destination or source register number
  - Constant: -2^{15} to $+2^{15} - 1$
  - Address: offset added to base address in rs
- Design Principle 4: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

MIPS Assembly to Machine code

- C Code

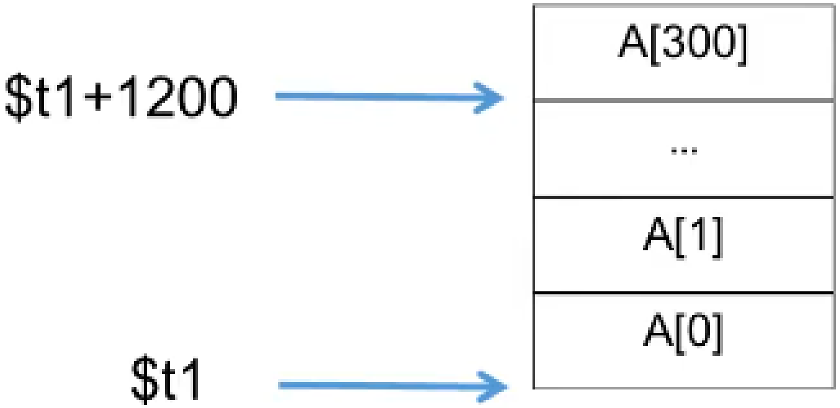
  A[300] = h + A[300];

- MIPS Assembly codes:

```
lw  $t0, 1200($t1) # Temporary reg $t0 gets A[300]
add $t0, $s2, $t0  # Temporary reg $t0 gets h + A[300]
sw  $t0, 1200($t1) # Stores h + A[300] back to A[300]
```

  ($t1 has the base of the array A, and $s2 corresponds to h)

- MIPS machine codes?



MIPS Assembly to Machine code

Register numbers
$t0 - $t7 are reg's 8 - 15
$t8 - $t9 are reg's 24 - 25
$s0 - $s7 are reg's 16 - 23

- A[300] = h + A[300];

```
lw  $t0, 1200($t1)  # Temporary reg $t0 gets A[300]
add $t0, $s2, $t0   # Temporary reg $t0 gets h + A[300]
sw  $t0, 1200($t1)  # Stores h + A[300] back into A[300]
```

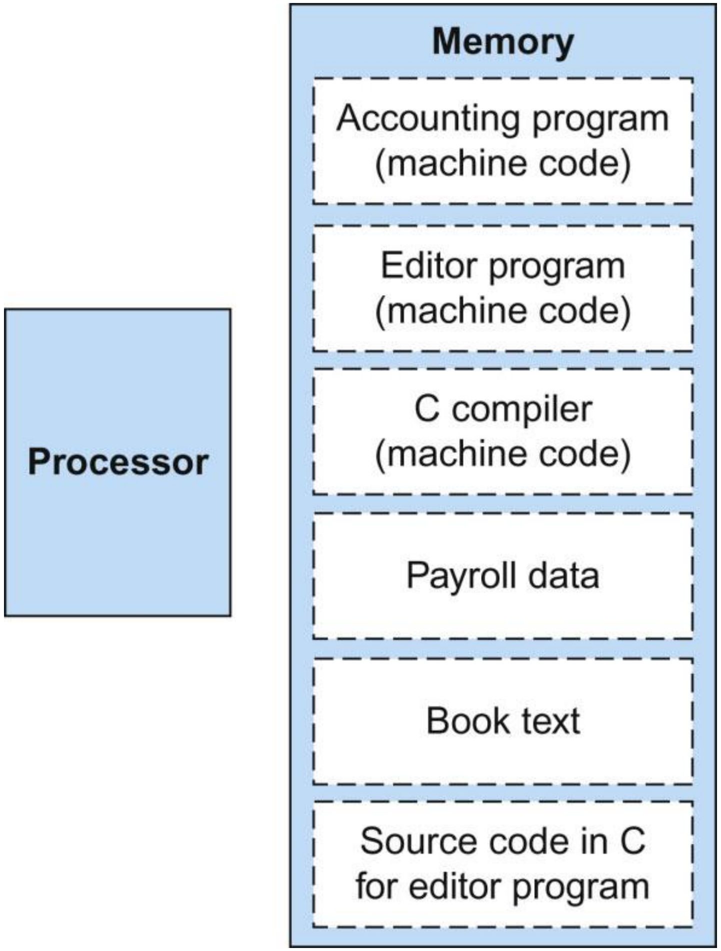
Instruction	Format	op	rs	rt	rd	shamt	funct	address
add	R	0	reg	reg	reg	0	32_{ten}	n.a.
sub (subtract)	R	0	reg	reg	reg	0	34_{ten}	n.a.
add immediate	I	8_{ten}	reg	reg	n.a.	n.a.	n.a.	constant
lw (load word)	I	35_{ten}	reg	reg	n.a.	n.a.	n.a.	address
sw (store word)	I	43_{ten}	reg	reg	n.a.	n.a.	n.a.	address

Op	rs	rt	rd	address /shamt	funct
35	9($t1)	8($t0)	1200		
0	18($s2)	8($t0)	8($t0)	0	32
43	9($t1)	8($t0)	1200		

C language
MIPS Assembly
Machine code
CPU data path
Digital circuit design
Transistors

Stored Program Computers

- 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


Processor
Memory
Accounting program (machine code)
Editor program (machine code)
C compiler (machine code)
Payroll data
Book text
Source code in C for editor program

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	`sll`
Shift right	>>	>>>	`srl`
Bitwise AND	&	&	`and, andi`
Bitwise OR	|	|	`or, ori`
Bitwise NOT	~	~	`nor`

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

Shift Operations

op	rs	rt	rd	**shamt**	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

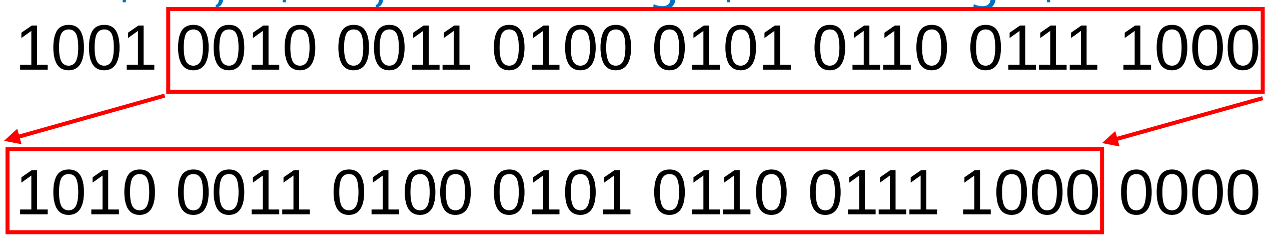
- shamt: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - **sll** by i bits multiplies by 2^i

左移

- Shift right logical
  - Shift right and fill with 0 bits
  - **srl** by i bits divides by 2^i (unsigned only)
- Shift right arithmetic
  - **sra**: shift right and fills with sign extension

Shift Instructions

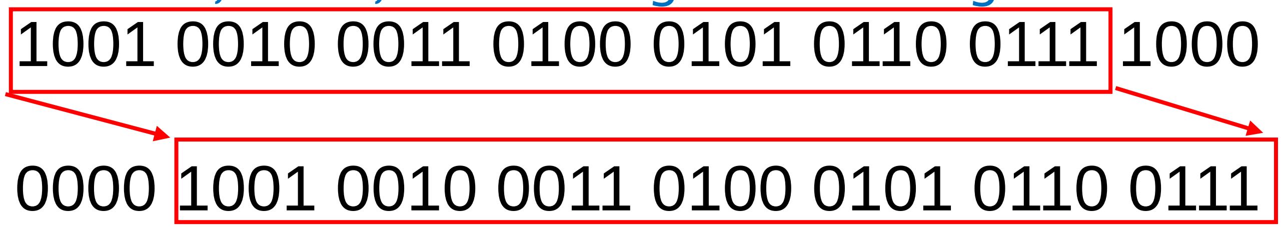
- `sll $t2, $s0, 4  # reg $t2 = reg $s0 << 4 bits`



1001 0010 0011 0100 0101 0110 0111 1000

1010 0011 0100 0101 0110 0111 1000 0000

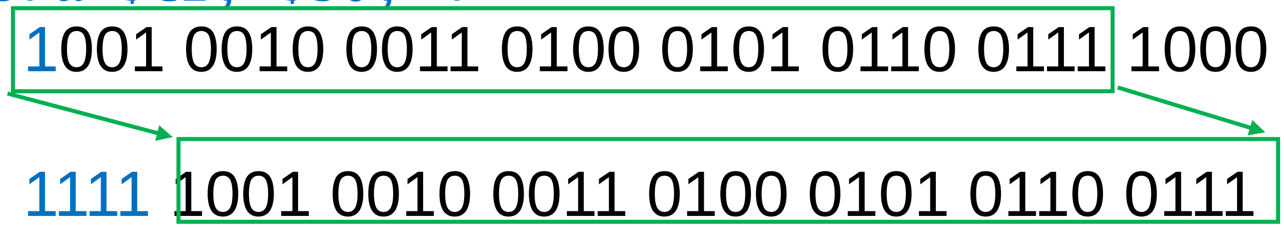
- `srl $t2, $s0, 4  # reg $t2 = reg $s0 >> 4 bits`



1001 0010 0011 0100 0101 0110 0111 1000

0000 1001 0010 0011 0100 0101 0110 0111

- `sra $t2, $s0, 4`



1001 0010 0011 0100 0101 0110 0111 1000

1111 1001 0010 0011 0100 0101 0110 0111

Logical Instructions

- AND:
  - *and $t0,$t1,$t2 # reg $t0 = reg $t1 & reg $t2*
- OR (NOR, XOR):
  - *or $t0, $t1,$t2 # reg $t0 = reg $t1 | reg $t2*
  - *nor $t1,$t1,$t3 # reg $t0 = ~ (reg $t1 | reg $t3)*
- How about NOT?
  - Can be implemented with NOR 3-operand instruction
  - a NOR b == NOT (a OR b)
  - *nor $t0, $t1, $zero*

Register 0: always read as zero

Conditional Operations

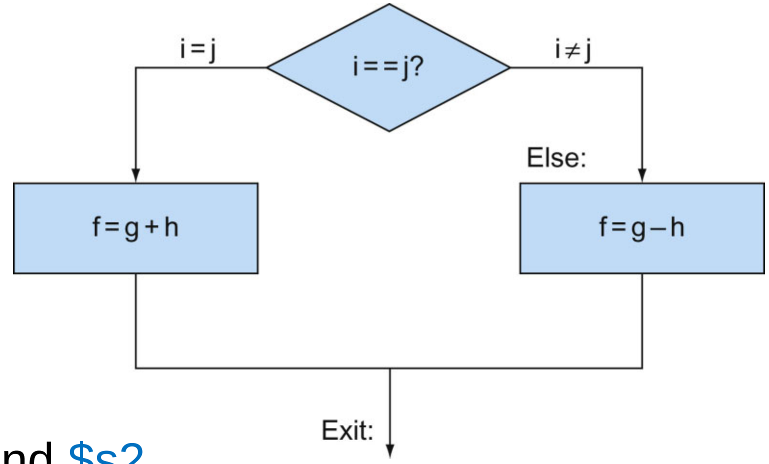
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- Conditional branch
  - *beq rs, rt, L1*

    if (rs == rt) branch to instruction labeled L1;
  - *bne rs, rt, L1*
    - if (rs != rt) branch to instruction labeled L1;
- Unconditional branch
  - *j L1*
    - unconditional jump to instruction labeled L1

Compiling If Statements

- C code

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



i and j are in $s3 and $s4,
f,g and h are in $s0, $s1 and $s2

- Compiled MIPS code:

```
      bne $s3, $s4, Else  # go to Else if i ≠ j
      add $s0, $s1, $s2   # f=g+h, skipped if i ≠ j
      j   Exit            # go to Exit
Else: sub $s0, $s1, $s2   # f=g-h, skipped if i = j
Exit:
```

Assembler calculates addresses

Compiling Loop Statements

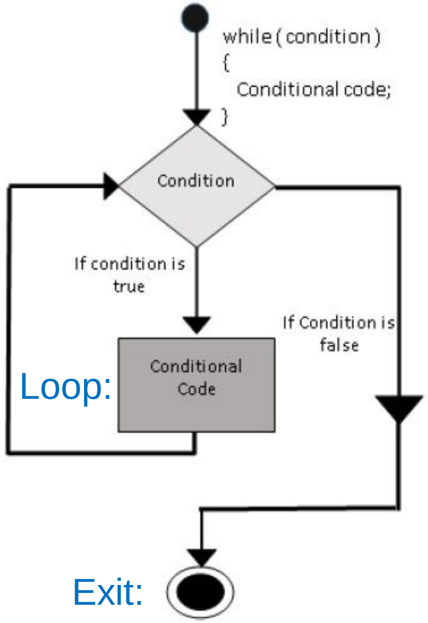
- C code:

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

- i in $s3, k in $s5, address of save in $s6

$s6+i*4	save[i]
	…
$s6	save[0]

左移代表乘法



- Compiled MIPS code:

```
Loop: sll   $t1, $s3, 2    # Temp reg $t1 = i * 4
      add   $t1, $t1, $s6  # $t1 = address of save[i]
      lw    $t0, 0($t1)    # Temp reg $t0 = save[i]
      bne   $t0, $s5, Exit # go to Exit if save[i]≠k
      addi  $s3, $s3, 1    # i = i + 1
      j     Loop           # go to Loop
Exit:
```

Summary of Design Principles

1: Simplicity favors regularity

- Keep all instructions the same size.

2: Smaller is faster

- Register vs memory
- Number of registers is small

3: Make the common case fast

- Immediate operand

4: Good design demands good compromises

- Keep formats as similar as possible

Summary of MIPS

- Instruction categories:
  - R-type
    - Arithmetic
    - Logical
  - I-type
    - Add immediate
    - Load/Store
  - J-type (we will learn it in lecture 4)
    - Jump and Branch

- Registers:
  - Operand
  - 32-32-bit register files
- Memory
  - Word
  - Memory Alignment
  - Big Endian

OP	$rs	$rt	$rd	shamt	funct

OP	$rs	$rt	immediate

OP	jump target