Instruction Set

Chixiao Chen

Overview

- First Question
- Review on MIPS Instruction Set
  - Courtesy by Prof. Huang from National Tsinghua Univ.
- RISC-V vs. MIPS
  - Key Difference
- An AI perspective

What Is Computer Architecture?

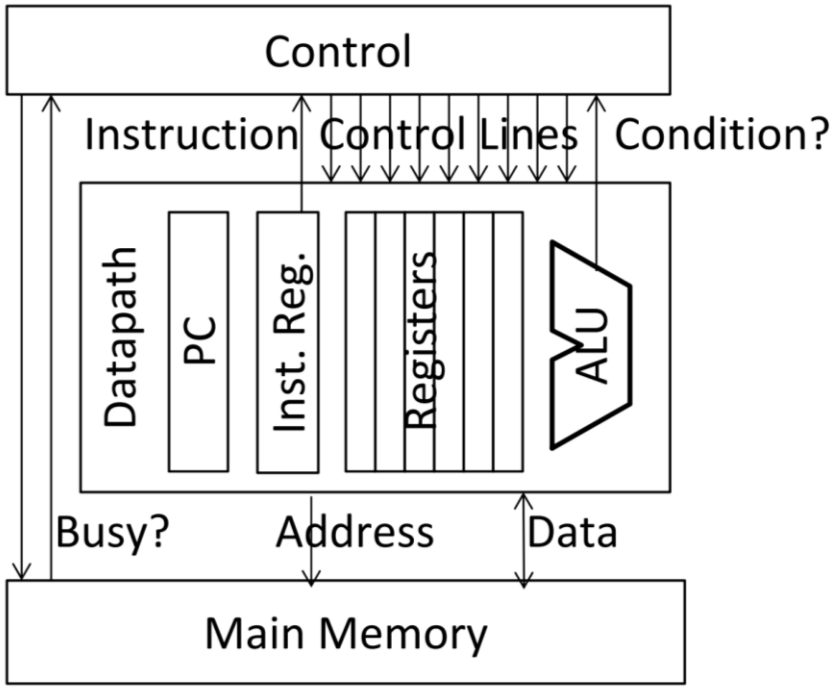
Computer Architecture =
Instruction Set Architecture
+ Machine Organization

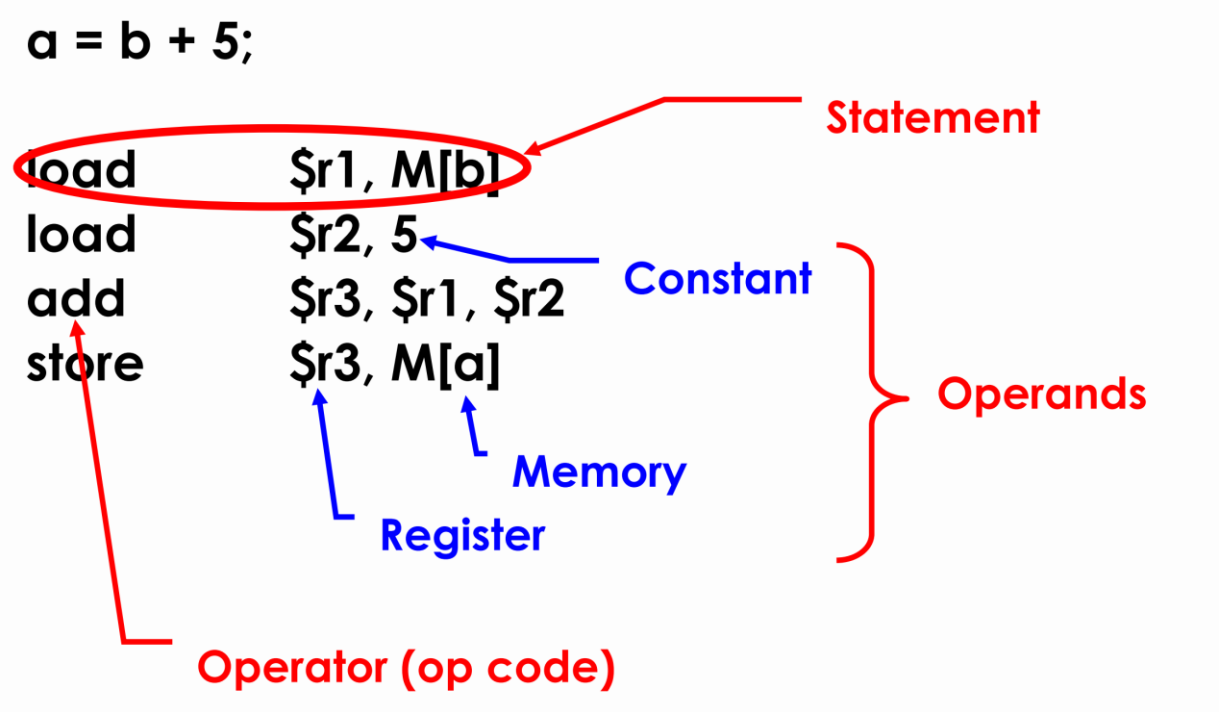
- **"... the attributes of a [computing] system as seen by the [assembly language] programmer, *i.e.* the conceptual structure and functional behavior ..."**

What are specified?

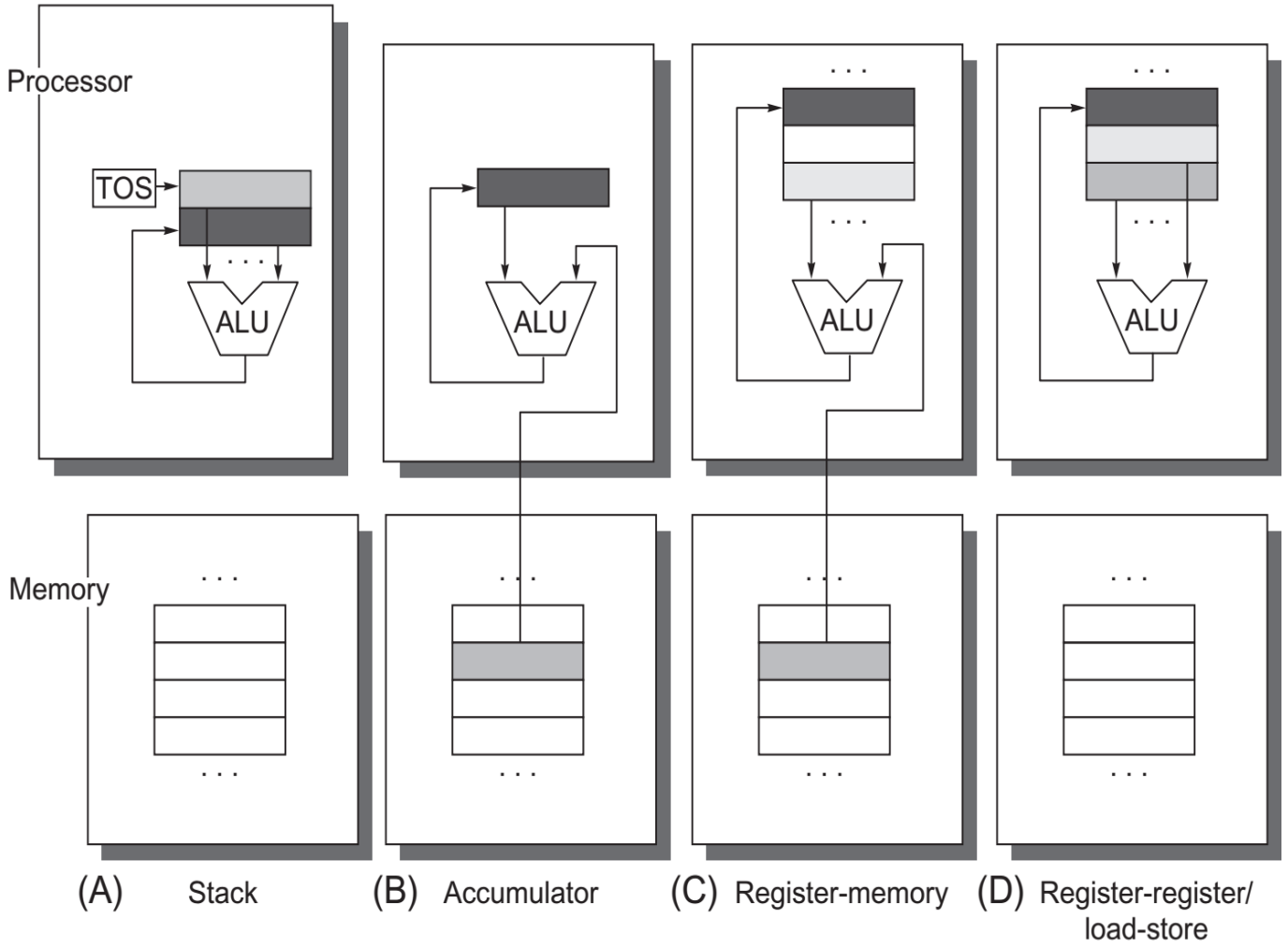
When you say a 8/16/32/64-bit processor…

- What does the most critical number mean?





Operand Locations



Stack	Accumulator
Push A	Load A
Push B	Add B
Add	Store C
Pop C	

Register (register-memory)	Register (load-store)
Load R1,A	Load R1,A
Add R3,R1,B	Load R2,B
Store R3,C	Add R3,R1,R2
	Store R3,C

Memory Addressing Type

Number of memory addresses	Maximum number of operands allowed	Type of architecture	Examples
0	3	Load-store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	IBM 360/370, Intel 80x86, Motorola 68000, TI TMS320C54x
2	2	Memory-memory	VAX (also has three-operand formats)
3	3	Memory-memory	VAX (also has two-operand formats)

Review on MIPS ISA

Operations of Hardware

- Syntax of basic MIPS arithmetic/logic instructions:

```
      1     2     3     4
   add $s0,$s1,$s2        # f = g + h
```

  1) operation by name
  2) operand getting result ("destination")
  3) 1st operand for operation ("source1")
  4) 2nd operand for operation ("source2")

- Each instruction is 32 bits
- Syntax is rigid: 1 operator, 3 operands
  - Why? Keep hardware simple via regularity
- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

MIPS Registers

- 32 registers, each is 32 bits wide
  - Why 32? *Design Principle 2:* smaller is faster
  - Groups of 32 bits called a *word* in MIPS
  - Registers are numbered from 0 to 31
  - Each can be referred to by number or name
  - Number references:

    `$0, $1, $2, … $30, $31`
  - By convention, each register also has a name to make it easier to code, e.g.,

    `$16 - $22` ➔ `$s0 - $s7` (C variables)

    `$8 - $15` ➔ `$t0 - $t7` (temporary)
- 32 x 32-bit FP registers (paired DP)
- Others: HI, LO, PC

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

Fig. 2.18

Data Transfer: Memory to Register (1/2)

- To transfer a word of data, need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a 1D array
    - Address it by supplying a pointer to a memory address
    - Offset (in bytes) from this pointer
    - The desired memory address is the sum of these two values, e.g., `8($t0)`
    - Specifies the memory address pointed to by the value in `$t0`, plus 8 bytes (why "bytes", not "words"?)
    - Each address is 32 bits

Data Transfer: Memory to Register (2/2)

- Load Instruction Syntax:

```
  1    2     3    4
lw $t0,12($s0)
```

  1) operation name
  2) register that will receive value
  3) numerical offset in bytes
  4) register containing pointer to memory

- Example: `lw $t0,12($s0)`
  - lw (Load Word, so a word (32 bits) is loaded at a time)
  - Take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`
- Notes:
  - `$s0` is called the *base register*, 12 is called the *offset*
  - Offset is generally used in accessing elements of array: base register points to the beginning of the array

Data Transfer: Register to Memory

- Also want to store value from a register into memory
- Store instruction syntax is identical to Load instruction syntax
- Example: `sw $t0,12($s0)`
  - sw (meaning Store Word, so 32 bits or one word are stored at a time)
  - This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into the memory address pointed to by the calculated sum

Addressing: Byte versus Word

- Every word in memory has an address, similar to an index in an array
- Early computers numbered words like C numbers elements of an array:
  - Memory[0], Memory[1], Memory[2], …

- Computers need to access 8-bit bytes as well as words (4 bytes/word)
- Today, machines address memory as bytes, hence word addresses differ by 4
  - `Memory[0], Memory[4], Memory[8], …`
  - This is also why lw and sw use bytes in offset

Memory Operand Example 2

- C code:

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

  - h in $s2, base address of A in $s3

- Compiled MIPS code:
  - Index 8 requires offset of A

```
lw  $t0, A($s3)    # load word
add $t0, $s2, $t0
sw  $t0, B($s3)    # store word
```

A = 32

B = 48

Constants

- Small constants used frequently (50% of operands)

  e.g., A = A + 5;
  B = B + 1;
  C = C - 18;

- Put 'typical constants' in memory and load them
- Constant data specified in an instruction:

```
addi $29, $29, 4
slti $8, $18, 10
andi $29, $29, 6
ori $29, $29, 4
```

- *Design Principle 3*: <u>Make the common case fast</u>

Immediate Operands

- Immediate: numerical *constants*
  - Often appear in code, so there are special instructions for them
  - Add Immediate:

```
f = g + 10          (in C)
addi $s0,$s1,10     (in MIPS)
```

    where `$s0,$s1` are associated with `f,g`
  - Syntax similar to `add` instruction, except that last argument is a number instead of a register
  - No subtract immediate instruction
    - Just use a negative constant

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

The Constant Zero

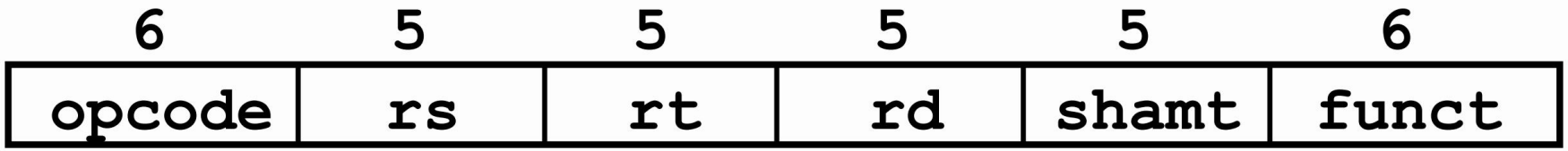
- The number zero (0), appears very often in code; so we define register zero
- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
  - This is defined in hardware, so an instruction like `addi $0,$0,5` will not do anything
- Useful for common operations
  - E.g., move between registers
    `add $t2, $s1, $zero`

MIPS Instruction Format

- One instruction is 32 bits
  => divide instruction word into "fields"
  - Each field tells computer something about instruction
- We could define different fields for each instruction, but MIPS is based on simplicity, so define 3 basic types of instruction formats:
  - *R-format*: for register
  - *I-format*: for immediate, and `lw` and `sw` (since the offset counts as an immediate)
  - *J-format*: for jump

R-Format Instructions (1/2)

- Define the following "fields":



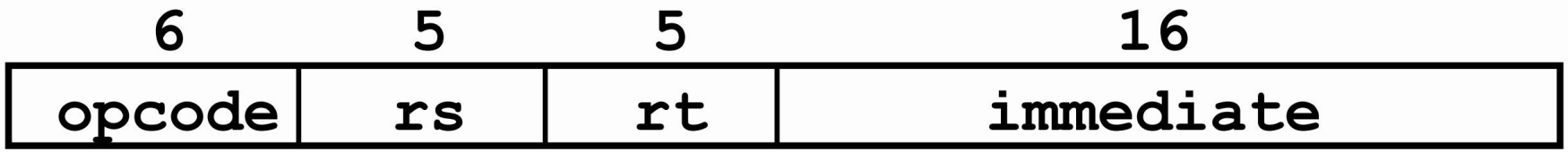
  - `opcode`: partially specifies what instruction it is (Note: 0 for all R-Format instructions)
  - `funct`: combined with `opcode` to specify the instruction
    Question: Why aren't `opcode` and `funct` a single 12-bit field?
  - `rs` (Source Register): *generally* used to specify register containing first operand
  - `rt` (Target Register): *generally* used to specify register containing second operand
  - `rd` (Destination Register): *generally* used to specify register which will receive result of computation

R-Format Instructions (2/2)

- Notes about register fields:
  - Each register field is exactly 5 bits, which means that it can specify any unsigned integer in the range 0-31. Each of these fields specifies one of the 32 registers by number.
- Final field:
  - `shamt`: contains the amount a shift instruction will shift by. Shifting a 32-bit word by more than 31 is useless, so this field is only 5 bits
  - This field is set to 0 in all but the shift instructions

I-Format Instructions

- Define the following "fields":



  - `opcode`: uniquely specifies an I-format instruction
  - `rs`: specifies the *only* register operand
  - `rt`: specifies register which will receive result of computation (*target register*)
  - `addi`, `slti`, immediate is sign-extended to 32 bits, and treated as a signed integer
  - 16 bits ➔ can be used to represent immediate up to 2^{16} different values

I-Format Example 1

- **MIPS Instruction:**

```
addi    $21,$22,-50
```

  - opcode = 8 (look up in table)
  - rs = 22 (register containing operand)
  - rt = 21 (target register)
  - immediate = -50 (by default, this is decimal)

decimal representation:

8	22	21	-50

binary representation:

001000	10110	10101	1111111111001110

I-Format Example 2

- **MIPS Instruction:**

```
lw    $t0,1200($t1)
```

  - opcode = 35 (look up in table)
  - rs = 9 (base register)
  - rt = 8 (destination register)
  - immediate = 1200 (offset)

decimal representation:

35	9	8	1200

binary representation:

100011	01001	01000	0000010010110000

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

NOT Operations

- Useful to invert bits in a word
  - Change 0 to 1, and 1 to 0
- MIPS has NOR 3-operand instruction
  - a NOR b == NOT (a OR b)

```
nor $t0, $t1, $zero
```

Register 0: always read as zero

$t1	0000 0000 0000 0000 0011 1100 0000 0000
$t0	1111 1111 1111 1111 1100 0011 1111 1111

MIPS Decision Instructions

```
beq   register1, register2, L1
```

- Decision instruction in MIPS:

  ```
  beq   register1, register2, L1
  ```

  "Branch if (registers are) equal"
  meaning :

  ```
  if  (register1==register2) goto L1
  ```

- Complementary MIPS decision instruction

  ```
  bne   register1, register2, L1
  ```

  "Branch if (registers are) not equal"
  meaning :

  ```
  if  (register1!=register2) goto L1
  ```

- These are called conditional branches

MIPS Goto Instruction

```
j     label
```

- MIPS has an unconditional branch:

  ```
  j     label
  ```

  - Called a Jump Instruction: jump directly to the given label without testing any condition
  - meaning :

    ```
    goto label
    ```

- Technically, it's the same as:

  ```
  beq     $0,$0,label
  ```

  since it always satisfies the condition
- It has the j-type instruction format

Compiling Loop Statements

- C code:

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

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

- Compiled MIPS code:

```
Loop:   sll  $t1, $s3, 2      #$t1=i x 4
        add  $t1, $t1, $s6    #$t1=addr of save[i]
        lw   $t0, 0($t1)      #$t0=save[i]
        bne  $t0, $s5, Exit   #if save[i]!=k goto Exit
        addi $s3, $s3, 1      #i=i+1
        j    Loop             #goto Loop
Exit: …
```

Inequalities in MIPS

- Until now, we've only tested equalities (== and != in C), but general programs need to test < and >
- Set on Less Than:
  - `slt rd, rs, rt`
    - if (rs < rt) rd = 1; else rd = 0;
  - `slti rt, rs, constant`
    - if (rs < constant) rt = 1; else rt = 0;

Compile by hand: `if (g < h) goto Less;`
Let `g: $s0, h: $s1`

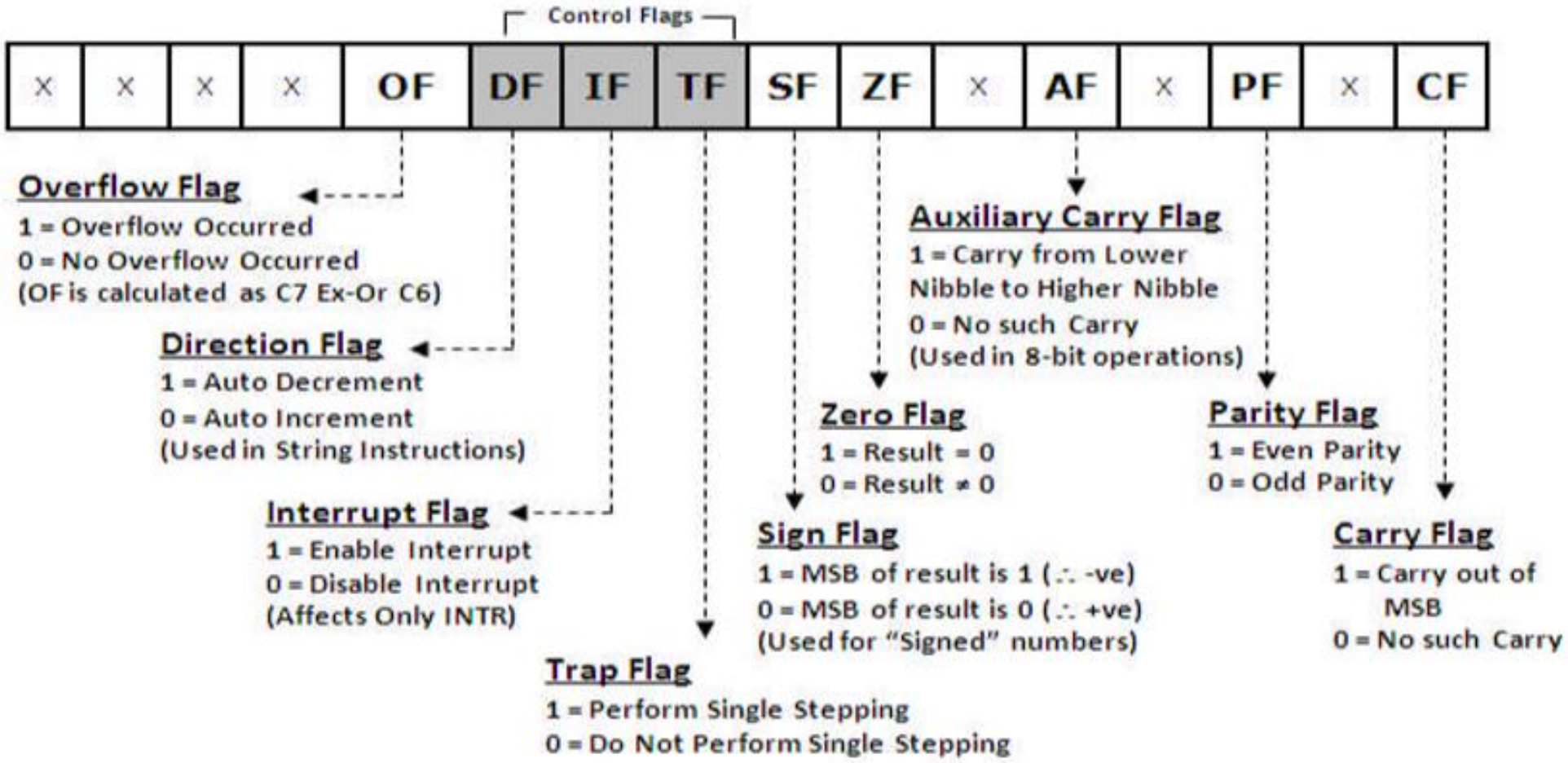
```
slt $t0,$s0,$s1    # $t0 = 1 if g<h
bne $t0,$0,Less    # goto Less if $t0!=0
```

- MIPS has no "branch on less than" => too complex

Different Branch Conditions Methods

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructionsbecause they pass information from one instruction to a branch
Condition register/ limited comparison	Alpha, MIPS	Tests arbitrary register with the result of a simple comparison (equality or zero tests)	Simple	Limited compare may affect critical path or require extra comparison for general condition
Compare and branch	PA-RISC, VAX, RISC-V	Compare is part of the branch. Fairly general compares are allowed (greater then, less then)	One instruction rather than two for a branch	May set critical path for branch instructions

Flag Register of 8086



Procedure Calling

- Steps required

Caller:

1. Place parameters in registers
2. Transfer control to procedure

Callee:

3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in register for caller
6. Return to place of call

C Function Call Bookkeeping

```
sum = leaf_example(a,b,c,d) . . .

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

- Return address `$ra`
- Procedure address `Labels`
- Arguments `$a0, $a1, $a2, $a3`
- Return value `$v0, $v1`
- Local variables `$s0, $s1, ..., $s7`

Note the use of register conventions

Registers Conventions for MIPS

0	zero	constant 0
1	at	reserved for assembler
2	v0	expression evaluation &
3	v1	function results
4	a0	arguments
5	a1	
6	a2	
7	a3	
8	t0	temporary: caller saves
. . .		(callee can clobber)
15	t7	
16	s0	callee saves
. . .		(caller can clobber)
23	s7	
24	t8	temporary (cont'd)
25	t9	
26	k0	reserved for OS kernel
27	k1	
28	gp	pointer to global area
29	sp	stack pointer
30	fp	frame pointer
31	ra	return address (HW)

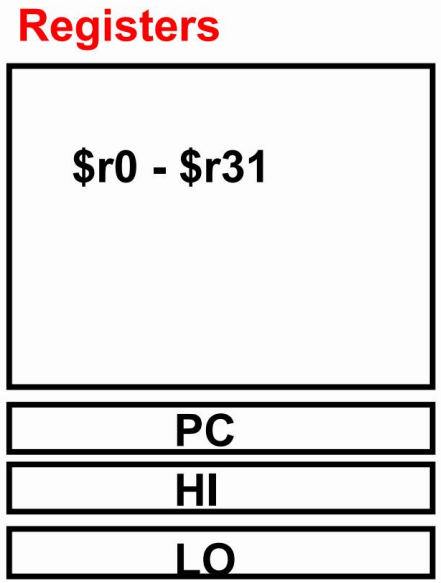
Fig. 2.18

Procedure Call Instructions

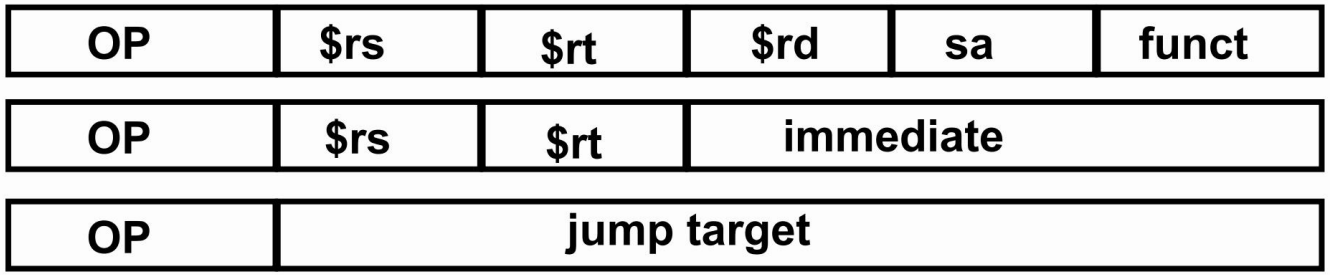
- Procedure call: jump and link
  `jal ProcedureLabel`
  - Address of following instruction put in $ra
  - Jumps to target address (i.e.,`ProcedureLabel`)
- Procedure return: jump register
  `jr $ra`
  - Copies $ra to program counter
  - Can also be used for computed jumps
    - e.g., for case/switch statements
      - Jump table is an array of addresses corresponding to labels in codes
      - Load appropriate entry to register
      - Jump register

MIPS ISA as an Example

- Instruction categories:
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide



RISC-V vs. MIPS

RISC-V Instruction Format

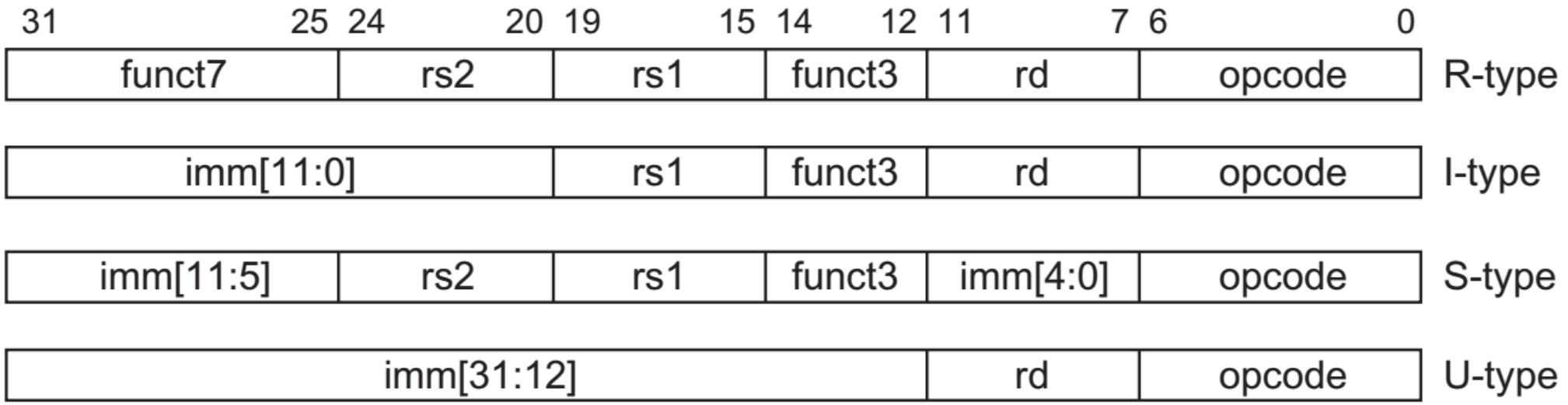


Figure A.23 The RISC-V instruction layout. There are two variations on these formata, called the SB and UJ formats; they deal with a slightly different treatment for immediate fields.

Why S-type? Is it different from I-type?

RISC-V Instruction Format

Instruction format	Primary use	rd	rs1	rs2	Immediate
R-type	Register-register ALU instructions	Destination	First source	Second source	
I-type	ALU immediates Load	Destination	First source base register		Value displacement
S-type	Store Compare and branch		Base register first source	Data source to store second source	Displacement offset
U-type	Jump and link Jump and link register	Register destination for return PC	Target address for jump and link register		Target address for jump and link

Extensions for RV

- RV supports extensions
- RV32I is the most common designs

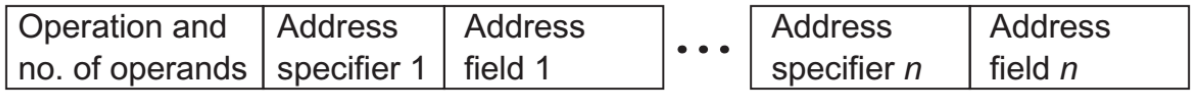
Name of base or extension	Functionality
RV32I	Base 32-bit integer instruction set with 32 registers
RV32E	Base 32-bit instruction set but with only 16 registers; intended for very low-end embedded applications
RV64I	Base 64-bit instruction set; all registers are 64-bits, and instructions to move 64-bit from/to the registers (LD and SD) are added
M	Adds integer multiply and divide instructions
A	Adds atomic instructions needed for concurrent processing; see Chapter 5
F	Adds single precision (32-bit) IEEE floating point, includes 32 32-bit floating point registers, instructions to load and store those registers and operate on them
D	Extends floating point to double precision, 64-bit, making the registers 64-bits, adding instructions to load, store, and operate on the registers
Q	Further extends floating point to add support for quad precision, adding 128-bit operations
L	Adds support for 64- and 128-bit decimal floating point for the IEEE standard
C	Defines a compressed version of the instruction set intended for small-memory-sized embedded applications. Defines 16-bit versions of common RV32I instructions
V	A future extension to support vector operations (see Chapter 4)
B	A future extension to support operations on bit fields
T	A future extension to support transactional memory
P	An extension to support packed SIMD instructions: see Chapter 4
RV128I	A future base instruction set providing a 128-bit address space

RISC-V Control flow Instructions

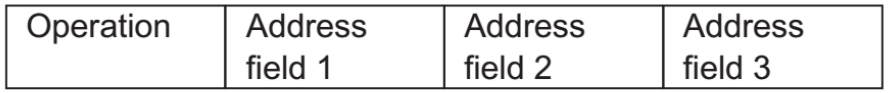
Instruction	Format	Meaning
`beq rs1, rs2, imm[12:1]`	SB	Branch if equal
`bne rs1, rs2, imm[12:1]`	SB	Branch if not equal
`blt rs1, rs2, imm[12:1]`	SB	Branch if less than, 2's complement
`bltu rs1, rs2, imm[12:1]`	SB	Branch if less than, unsigned
`bge rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, 2's complement
`bgeu rs1, rs2, imm[12:1]`	SB	Branch if greater or equal, unsigned
`jal rd, imm[20:1]`	UJ	Jump and link
`jalr rd, rs1, imm[11:0]`	I	Jump and link register

Table 3.4: Listing of RV32I control transfer instructions.

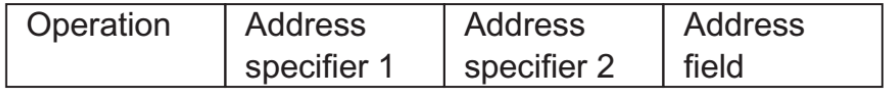
Instruction format with Variable Length

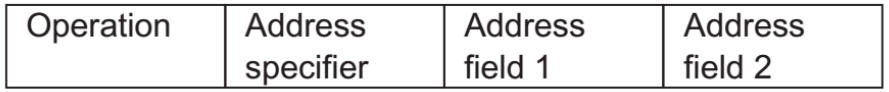


(A) Variable (e.g., Intel 80x86, VAX)



(B) Fixed (e.g., RISC V, ARM, MIPS, PowerPC, SPARC)





(C) Hybrid (e.g., RISC V Compressed (RV32IC), IBM 360/370, microMIPS, Arm Thumb2)

Instruction Compression in RV

15 14 13	12	11 10 9 8 7	6 5 4 3 2	1 0	
000	nzuimm[5]	rs1/rd≠0	nzuimm[4:0]	10	C.SLLI *(HINT, rd=0; RV32 NSE, nzuimm[5]=1)*
000	0	rs1/rd≠0	0	10	C.SLLI64 *(RV128; RV32/64 HINT; HINT, rd=0)*
001	uimm[5]	rd	uimm[4:3|8:6]	10	C.FLDSP *(RV32/64)*
001	uimm[5]	rd≠0	uimm[4|9:6]	10	C.LQSP *(RV128; RES, rd=0)*
010	uimm[5]	rd≠0	uimm[4:2|7:6]	10	C.LWSP *(RES, rd=0)*
011	uimm[5]	rd	uimm[4:2|7:6]	10	C.FLWSP *(RV32)*
011	uimm[5]	rd≠0	uimm[4:3|8:6]	10	C.LDSP *(RV64/128; RES, rd=0)*
100	0	rs1≠0	0	10	C.JR *(RES, rs1=0)*
100	0	rd≠0	rs2≠0	10	C.MV *(HINT, rd=0)*
100	1	0	0	10	C.EBREAK
100	1	rs1≠0	0	10	C.JALR
100	1	rs1/rd≠0	rs2≠0	10	C.ADD *(HINT, rd=0)*
101	uimm[5:3|8:6]		rs2	10	C.FSDSP *(RV32/64)*
101	uimm[5:4|9:6]		rs2	10	C.SQSP *(RV128)*
110	uimm[5:2|7:6]		rs2	10	C.SWSP
111	uimm[5:2|7:6]		rs2	10	C.FSWSP *(RV32)*
111	uimm[5:3|8:6]		rs2	10	C.SDSP *(RV64/128)*

Compression saves 25% code volume

Instructions for AI?

Principle Reviews

- Rule No.1
  - **Simplicity favors regularity**
- Rule No.2
  - **Smaller is faster**
- Rule No.3
  - **Make the common case fast!**

Cambricon 寒武纪

Instruction Type		Examples	Operands
Control		jump, conditional branch	register (scalar value), immediate
Data Transfer	Matrix	matrix load/store/move	register (matrix address/size, scalar value), immediate
	Vector	vector load/store/move	register (vector address/size, scalar value), immediate
	Scalar	scalar load/store/move	register (scalar value), immediate
Computational	Matrix	matrix multiply vector, vector multiply matrix, matrix multiply scalar, outer product, matrix add matrix, matrix subtract matrix	register (matrix/vector address/size, s-calar value)
	Vector	vector elementary arithmetics (add, subtract, multiply, divide), vector transcendental functions (exponential, logarithmic), dot product, random vector generator, maximum/minimum of a vector	register (vector address/size, scalar value)
	Scalar	scalar elementary arithmetics, scalar transcendental functions	register (scalar value), immediate
Logical	Vector	vector compare (greater than, equal), vector logical operations (and, or, inverter), vector greater than merge	register (vector address/size, scalar)
	Scalar	scalar compare, scalar logical operations	register (scalar), immediate

Thanks