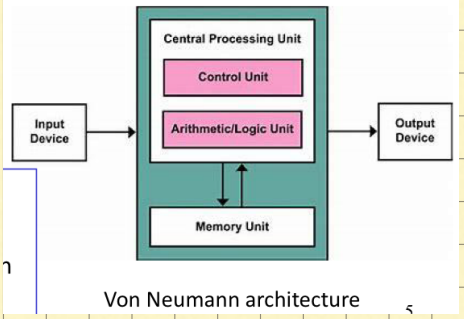
Instruction Set

- To command a computer's hardware, you must speak its language
  - ✓ Instructions: words of a computer's language
  - ✓ Instruction set: vocabulary of commands
- Two forms of instruction set:
  - ✓ Assembly language: written by people
  - ✓ Machine language: read by computer
- A program (in say, C) is compiled into an executable program that is composed of machine instructions
  - ✓ This executable program must also run on future machines
  - ✓ Each Intel processor reads in the same x86 instructions, but each processor handles instructions differently
- Java programs are converted into portable bytecode that is converted into machine instructions during execution (just-in-time compilation)

- Instruction set of different machine are similar, because
  - ✓ they base on similar design principles
  - ✓ several basic operations are provided
  - ✓ computer designers have a common goal
- Design target:
  - ✓ easy to build the hardware and compiler
  - ✓ maximizing performance and minimizing cost and energy
- Important design principles:
  - ✓ keep the hardware simple – the chip must only implement basic primitives and run fast
  - ✓ keep the instructions regular – simplifies the decoding/ scheduling of instructions

x86指令有的长有的短.

Von Neumann Architecture



Von Neumann architecture

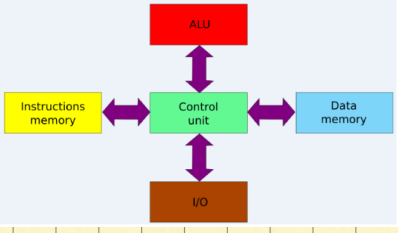
Stored-program computer:

instructions and data of many types can be stored in memory as numbers

指令与数据以二进制形式存于一处

Harvard Architecture

- Two separated memories:
  - ✓ data memory
  - ✓ instructions memory
- Two separated buses: data bus and memory bus
- More efficient than von Neumann, widely used in embedded systems



哈佛结构的指令和数据分开存.
因此易于并行
常用于嵌入式系统中.

Instruction Set

- All instruction set are similar
  - ✓ Once learn one, easy to pick up others
- We will use MIPS as an example
  - ✓ MIPS: Microprocessor without interlocked pipeline stages
  - ✓ History of MIPS
- RISC vs CISC
  - ✓ RISC: reduced instruction set computer, e.g. MIPS, ARM, PowerPC, RISC-V
  - ✓ CISC: complex instruction set computer, e.g. x86

C code a = b + c + d + e;

translates into the following assembly code:

```
add  a, b, c          add  a, b, c
add  a, a, d    or    add  f, d, e
add  a, a, e          add  a, a, f
```

- Instructions are simple: fixed number of operands (unlike C)
- A single line of C code is converted into multiple lines of assembly code
- Some sequences are better than others… the second sequence needs one more (temporary) variable f

C code f = (g + h) – (i + j);

translates into the following assembly code:

```
add  t0, g, h         add  f, g, h
add  t1, i, j   or    sub  f, f, i
sub  f,  t0, t1       sub  f, f, j
```

- Each version may produce a different result because floating-point operations are not necessarily associative and commutative… more on this later

右边多用了一个寄存器，但并行效果好

浮点数不满足交换律和结合律

Design Principle 1：简单源于规整

规整指令

- Simplicity favors regularity
  - ✓ Regularity makes implementation simpler
  - ✓ Simplicity enables higher performance at lower cost

Operands

- In C, each "variable" is a location in memory
- In hardware, each memory access is expensive – if variable *a* is accessed repeatedly, it helps to bring the variable into an on-chip scratchpad and operate on the scratchpad (registers)
- To simplify the instructions, we require that each instruction (add, sub) only operate on registers
- Note: the number of operands (variables) in a C program is very large; the number of operands in assembly is fixed… there can be only so limited number of registers

C语言中，变量都存在内存里。

但硬件中，访问内存开销较大，一般对寄存器做操作。

汇编中指令只能对寄存器做操作。

（除了lw、sw）

Register

因为x86中有指令能直接对内存操作，故能省寄存器。

- The MIPS ISA has 32 registers (x86 has 8 registers) – Why not more? Why not less?
- Each register is 32-bit wide (modern 64-bit architectures have 64-bit wide registers)
- A 32-bit entity (4 bytes) is referred to as a word
- To make the code more readable, registers are partitioned as $s0-$s7 (C/Java variables), $t0-$t9 (temporary variables)…

太多寄存器会造成浪费，而且会挤占空间

太少寄存器就需要经常从内存搬数据

寄存器宽度取决于机器

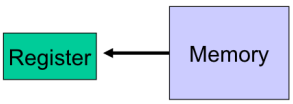
1 word = 4 byte = 32 bit (32位)

= 8 byte = 64 bit (64位)

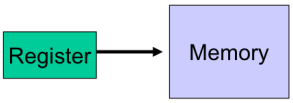
Memory Operands

- Values must be fetched from memory before (add and sub) instructions can operate on them

Load word
lw $t0, memory-address



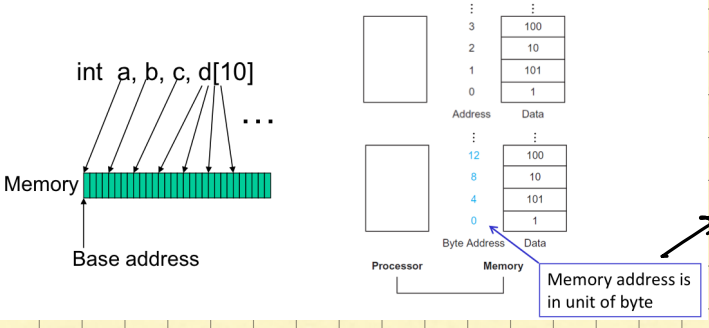
Store word
sw $t0, memory-address



How is memory-address determined?

Memory Address

- The compiler organizes data in memory… it knows the location of every variable (saved in a table)… it can fill in the appropriate mem-address for load-store instructions



寻址：基址+偏移量

Memory基于byte

Immediate Operands

立刻操作数：输入是一个常量

- An instruction may require a constant as input

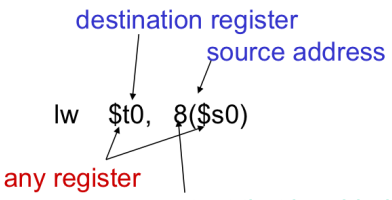
- An immediate instruction uses a constant number as one of the inputs (instead of a register operand)

```
addi   $s0, $zero, 1000   # the program has base address
                          #  1000 and this is saved in $s0
                          # $zero is a register that always
                          # equals zero
addi   $s1, $s0, 0        # this is the address of variable a
addi   $s2, $s0, 4        # this is the address of variable b
addi   $s3, $s0, 8        # this is the address of variable c
addi   $s4, $s0, 12       # this is the address of variable d[0]
```

	1000	1004	1008	1012
地址	a	b	c	d

Memory Instruction Format

- The format of a load instruction:



$s0获取寄存器s0的内容

($s0)代表基址，前面有8代表基址加8.

Convert to assembly:

C code: d[3] = d[2] + a;

```
Assembly:  # addi instructions as before
           lw    $t0, 8($s4)   #  d[2] is brought into $t0
           lw    $t1, 0($s1)   #   a  is brought into $t1
           add   $t0, $t0, $t1 #  the sum is in $t0
           sw    $t0, 12($s4)  #  $t0 is stored into d[3]
```

Assembly version of the code continues to expand!

Register 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!

寄存器访问快于内存

尽量用register存变量

Design Principle 2&3

- *Design Principle 2*: Smaller is faster
  - Register vs. memory
  - Number of registers is small
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

越小越快

要令一般情况快

尽量用立即指令而非load指令

Numeric Representation

- Decimal 35_{10}
- Binary 00100011_2
- Hexadecimal (compact representation)
  0x 23 or 23_{hex}

  0-15 (decimal) → 0-9, a-f (hex)

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
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0011_2$
    $= 0 + \ldots + 0\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 0 + 0 + 2 + 1 = 3_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2's Complement Signed Integers

- Given an n-bit number, define the value as follows:

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

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^2 + 0\times2^1 + 0\times2^0$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

Signed Negation

- 2's complement = 1's complement + 1
  - 1's complement means 1 → 0, 0 → 1
  - Using $\bar{x}$ represent 1's complement

$$x + \bar{x} = 1111\ldots111_2 = -1$$
$$\bar{x} + 1 = -x$$

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

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
  - E.g. we copy 8-bit to register and then want to extend it to be 16-bit or 32-bit
- 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

都先对输入的数补位再放到目标处

正数前面补0，负数补1

Instruction Formats

Instructions are represented as 32-bit numbers (one word), broken into 6 fields

Register-type

R-type instruction				add $t0, $s1, $s2	
000000	10001	10010	01000	00000	100000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

Immediate-type

I-type instruction			lw $t0, 32($s3)
6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

指令用32 bit机器码表达

分成6部分，分别占bit数 6 5 5 5 5 6

opcode表示指令类型，funct表示操作类型

rs、rt表示源寄存器，rd是目标寄存器

shamt表示移位.

R-type的opcode是全0.

I-type的opcode代表什么操作

通过前6 bit可以区分R、I、J-type.

Stored-Program Computer

- Today's computers are built on two key principles:
  - Instructions are represented as numbers.
  - Programs are stored in memory to be read or written, just like data.
- These principles lead to the stored-program concept.

电脑中的指令都由数字表示

程序与数据类似，都存在内存中，可读写.

Design Principle 4

- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

好的设计取决于好妥协.

尽量使格式类似

Logical Operations

Logical ops	C operators	Java operators	MIPS instr
Shift Left	<<	<<	sll
Shift Right	>>	>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

右移：srl 1000...01 → 0100...00 (shift right logically)

sra 1000...01 → 1100...00 (shift right arithmetically)

左移没有sla

Control Instructions

- Conditional branch: Jump to instruction L1 if register1 equals register2: `beq register1, register2, L1`
  Similarly, bne and slt (set-on-less-than)

- Unconditional branch:
  `j L1` → 跳转到 label
  `jr $s0` → 跳转到 s0 内部数字表示的地址

Convert to assembly:

```
if (i == j)                    bne  $s3, $s4, Else
   f = g+h;                    add  $s0, $s1, $s2
else                           j    Exit
   f = g-h;              Else: sub  $s0, $s1, $s2
                         Exit:
```

- How to compile:
  - If (a < b) …, else, …
- slt rd, rs, rt set less than
  - if (rs < rt) rd = 1; else rd = 0;
- slti rt, rs, constant
  - if (rs < constant) rt = 1; else rt = 0;
- Use in combination with beq, bne

```
slt $t0, $s1, $s2  # if ($s1 < $s2)
bne $t0, $zero, L  #  branch to L
```

Convert to assembly:

```
Convert to assembly:
  if (i < j)
     f = g+h;
  else
     f = g-h;
```

```
        slt  $t0, $s3, $s4
        beq  $t0, $zero, Else
        add  $s0, $s1, $s2
        j    Exit
Else:   sub  $s0, $s1, $s2
Exit:
```

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

Loop

Convert to assembly:

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

i and k are in $s3 and $s5 and
base of array save[] is in $s6

```
Loop:  sll   $t1, $s3, 2
       add   $t1, $t1, $s6
       lw    $t0, 0($t1)
       bne   $t0, $s5, Exit
       addi  $s3, $s3, 1
       j     Loop
Exit:
```

→ 左移2位，相当于×4.
→ 表示每次数组向后移一个.
每个整数占1 byte，相当于8 bit

Pseudo Instructions

- blt $s0, $s1, Label ⟺ slt $s2, $s0, $s1
  bne $s2, $zero, Label
  - If s0<s1, jump to Label
- bgt $s0, $s1, Label
  - If s0<s1, jump to Label
- ble $s0, $s1, Label
  - If s0<=s1, jump to Label
- beqz $s0, Label
  - If s0==0, jump to Label
- li $t0, 5
  - Load immediate, t0 = 5
- Move $t0, $s0
  - t0 = s0

There is no such instructions in hardware, The assembler translates them into a combination of real instructions

左边都是伪指令，即硬件中并没有这些指令，需汇编器将其翻译成真指令.

更多的决策语句能使代码易读易写，但更少的决策语句可在执行时简化任务.
更少的代码并不意味着执行的指令更少.

Branch Instruction Design

- Why not blt, bge, etc?
- Hardware for <, ≥, … slower than =, ≠
  - Combining with branch involves more work per instruction, requiring a slower clock
  - All instructions penalized!
- beq and bne are the common case
- This is a good design compromise

尽量要使单个指令的时钟周期较短，若某种指令过长的话，就将其拆成两个更好。
硬件中 ≤、≥要比 =、≠用时长，且 =、≠较常用。
故指令中有 beq、bne 而无 blt、bge。

Signed vs. Unsigned

- Signed comparison: slt, slti
- Unsigned comparison: sltu, sltui
- Example
  - $s0 = 1111 1111 1111 1111 1111 1111 1111 1111
  - $s1 = 0000 0000 0000 0000 0000 0000 0000 0001
  - slt $t0, $s0, $s1 # signed
    - –1 < +1 ⇒ $t0 = 1
  - sltu $t0, $s0, $s1 # unsigned
    - +4,294,967,295 > +1 ⇒ $t0 = 0

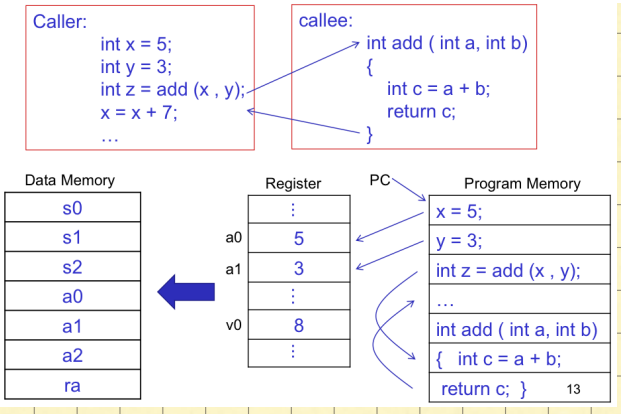
The register contains bits without meaning.

Are the bits represents a signed number or unsigned one? See the instruction!

是不是有符号取决于用什么指令。

Procedures

- A procedure or function is one tool used by the programmers to structure programs
  - Benefit: easy to understand, reuse code
- We can think of a procedure like a spy
  - acquires resources → performs task → covers his tracks → returns back with desired result
- When the procedure is executed (when the caller calls the callee), there are six steps
  - parameters (arguments) are placed where the callee can see them
  - control is transferred to the callee
  - acquire storage resources for callee
  - execute the procedure
  - place result value where caller can access it
  - return control to caller



Register Used during Procedure Calling

- The registers are used to hold data between the caller and the callee
  - $a0 - $a3: four argument registers to pass parameters
  - $v0 - $v1: two value registers to return the values
  - $ra: one return address register to return to the point of origin in the caller

Jump and Link

- *program counter* (PC)
  - A special register maintains the address of the instruction currently being executed
- The procedure call is executed by invoking the jump-and-link (jal) instruction – the current PC (actually, PC+4) is saved in the register $ra and we jump to the procedure's address (the PC is accordingly set to this address)

```
jal    NewProcedureAddress
```

- Since jal may over-write a relevant value in $ra, it must be saved somewhere (in memory?) before invoking the jal instruction
- How do we return control back to the caller after completing the callee procedure?

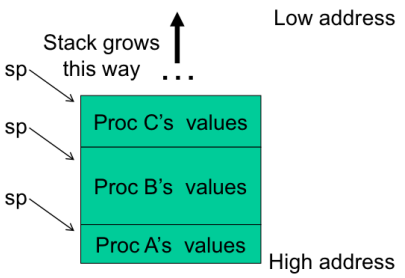
jal 做了两件事，一个是跳转，另一个是将 ra 的值修改成跳转前下一行的地址。

Registers

- The 32 MIPS registers are partitioned as follows:
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address

The Stack

The registers for a procedure are volatile, it disappears every time we switch procedures. Therefore, a procedure's values in the registers are backed up in memory on a stack




```
Proc  A
        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

每次调用就压栈，结束调用就出栈。

Storage Management on a Call/Return

- A new procedure must create space for all its variables on the stack
- Before executing the jal, the caller must save relevant values in $s0-$s7, $a0-$a3, $ra, temps into its own stack space
- Arguments are copied into $a0-$a3; the jal is executed
- After the callee creates stack space, it updates the value of $sp
- Once the callee finishes, it copies the return value into $v0, frees up stack space, and $sp is incremented
- On return, the caller may bring in its stack values, ra, temps into registers
- The responsibility for copies between stack and registers may fall upon either the caller or the callee

Example: Leaf Procedure

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

```
leaf_example:
   addi   $sp, $sp, -12
   sw     $t1, 8($sp)
   sw     $t0, 4($sp)
   sw     $s0, 0($sp)
   add    $t0, $a0, $a1
   add    $t1, $a2, $a3
   sub    $s0, $t0, $t1
   add    $v0, $s0, $zero
   lw     $s0, 0($sp)
   lw     $t0, 4($sp)
   lw     $t1, 8($sp)
   addi   $sp, $sp, 12
   jr     $ra
```

先保存原来的值

将原来的值移回去.

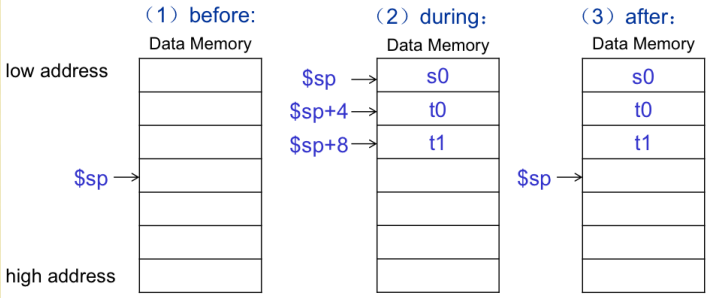
Save t0,t1,s0
Protect environment

Procedure body

Restore t0 t1 s0

Return result

The caller has saved:
g→$a0,
h→$a1,
i→$a2,
j→$a3,
return address → $ra



To avoid too many memory operations:
$t0 - $t9: temporary registers are not preserved by the callee
$s0 - $s7: saved registers must be preserved by the callee if used

t0~t9 不一定要保护
s0~s7 一定要保护

Example: non-leaf procedure

- 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

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

```
fact:
   addi   $sp, $sp, -8
   sw     $ra, 4($sp)
   sw     $a0, 0($sp)
   slti   $t0, $a0, 1
   beq    $t0, $zero, L1
   addi   $v0, $zero, 1
   addi   $sp, $sp, 8
   jr     $ra
L1:
   addi   $a0, $a0, -1
   jal    fact
   lw     $a0, 0($sp)
   lw     $ra, 4($sp)
   addi   $sp, $sp, 8
   mul    $v0, $a0, $v0
   jr     $ra
```

保存3个

Compare n<1

Return 1

Fact(n-1)

Return n*fact(n-1)

Notes:
The caller saves $a0 and $ra in its stack space.
Temps are never saved.

Saving Conventions

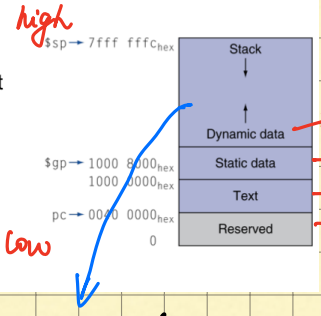
- Caller saved: Temp registers $t0-$t9 (the callee won't bother saving these, so save them if you care), $ra (it's about to get over-written), $a0-$a3 (so you can put in new arguments)
- Callee saved: $s0-$s7 (these typically contain "valuable" data)

调用者保护: $t0~$t9, $ra, $a0~$a3

被调用者保护: $s0~$s7.

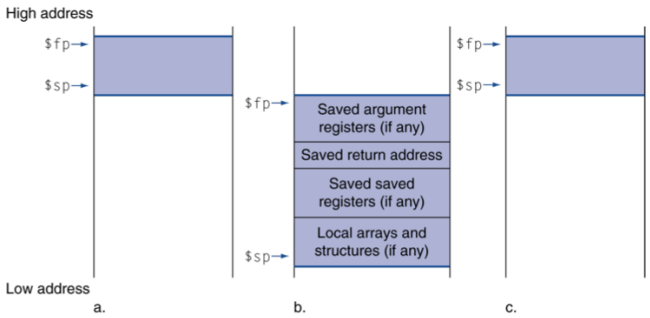
Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



heap. 上下不断调整
固定数据
程序
电脑系统区，不能用

Local Data on the Stack



$fp基本上不动，$sp上下不断动

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

MIPS Addressing

- Addressing: how the instructions identify the operands of the instruction.
- MIPS Addressing mode:
  - Immediate addressing — addi $s0, $s1, 5
  - Register addressing — add $s0, $s1, $s2
  - Base/Displacement addressing — lw $s0, 0($s1)
  - PC-relative addressing — bne $s0, $s1, EXIT
  - Pseudo-direct addressing — j EXIT

MIPS中有5种寻址方式.

Immediate Addressing

- For instructions including immediate
  - E.g. addi, subi, andi, ori

op	rs	rt	immediate
6 bits	5 bits	5 bits	16 bits

立即数只有16 bit.

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant: y=x+4000000

这个数过大

4000000_{dec} = 11 1101 0000 1001 0000 0000_{bin}

lui rt, constant — load upper immediate

  - Copies 16-bit constant to left 16 bits of rt
  - Clears right 16 bits of rt to 0

lui $s0, 61 → 0000 0000 0011 1101 0000 0000 0000 0000

ori $s0, $s0, 2304 → 0000 0000 0011 1101 0000 1001 0000 0000

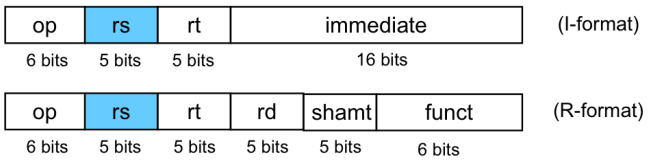
61就是高16位
2304就是低16位.

对比较大的数，就先用lui将数的前面位load到rt中，再用ori将rt与后16位作操作最终rt中就有结果

Register Addressing

使用寄存器做为操作数.

- Using register as the operand
- E.g. add, addi, sub, subi, lw, …



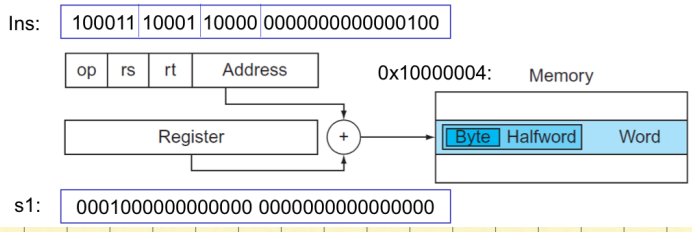
Base/Displacement Addressing

基址+偏移量

- the operand is at the memory, whose address is the sum of a register and a constant lw/lh/lb/sw/sh/sb
  - e.g. lw $s0, 4($s1)

    op of lw: 100011

    rs: 10001 (address of s1), rt: 10000 (address of s0), address: 100 (4)



Branch Addressing (PC-relative addressing)

只能在PC当前位置前后一定范围内跳转

(-128K, 128K)(bit) $2^{10}=1K$

指令中16位有符号. $2^5=32$ 然后还要×4. 故能前后 128K bit.

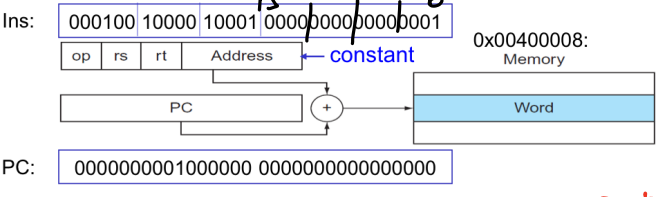
- Branch instructions specify
  - Opcode, two registers, target address
  - e.g. beq $s0 $s1 label

beq $s0 $s1 label 0x00400000

add $s2 $s3 $s4

lable: sub $s2 $s3 $s4 0x00400008

- Most branch targets are near branch
  - Forward or backward



- PC-relative addressing
  - Target address = PC + 4 + constant × 4

下条指令的位置

下条指令距目标指令行数,左移2位

7

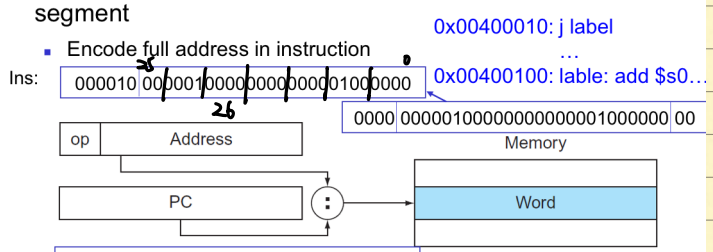
- Loop code from earlier example
  - Assume Loop at location 80000

```
Loop: sll  $t1, $s3, 2      80000
      add  $t1, $t1, $s6    80004
      lw   $t0, 0($t1)      80008
      bne  $t0, $s5, Exit   80012
      addi $s3, $s3, 1      80016
      j    Loop             80020
Exit: …                     80024
```

0	0	19	9	4	0	
0	9	22	9	0	32	
35	9	8	0			
5	8	21	2			
8	19	19	1			
2	20000					

Jump Addressing (Pseudo-direct addressing)

- Jump (`j` and `jal`) targets could be anywhere in text segment
  - Encode full address in instruction



- (Pseudo)Direct jump addressing
  - Target address = PC31…28 : (address × 4)

左移2位

保留PC前4位 拼接

目标地址32bit的前4位和后2位删除存在指令中，执行时先左移2位再拼接到PC的前4位后。

范围 (-256M, 256M) (bit)

指令中26bit 左移两位后有28 bit. $2^{20} = 1M$

$2^8 = 256$. 取绝前后256M bit.

Branching Far Away

- If branch target is too far to encode with 16-bit offset, assembler rewrites the code
- Example

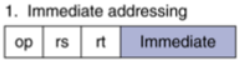
```
beq $s0,$s1, L1
        ↓
bne $s0,$s1, L2
j L1
L2:  ...
```

汇编器会重写beq转分支

避免跳转太远

Addressing Mode

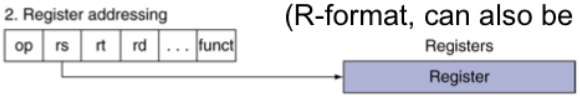
1. Immediate addressing



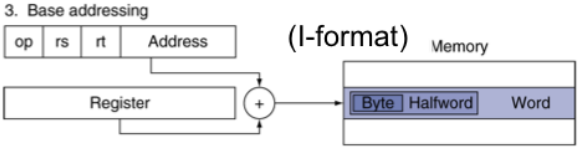
(I-format instruction)

2. Register addressing

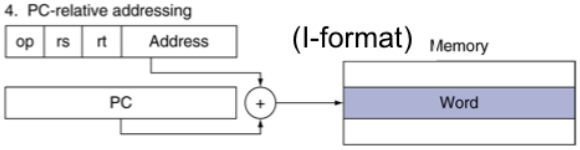
(R-format, can also be I-format)



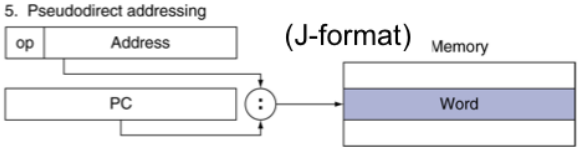
3. Base addressing



4. PC-relative addressing

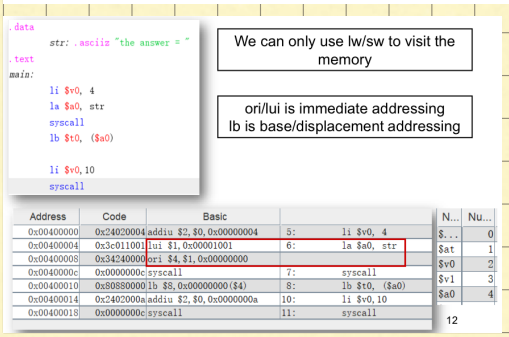


5. Pseudodirect addressing



32bit 地址怎么找 → Addressing

32bit 指令怎么分 → Format



MIPS没有直接寻址的方法.

(direct addressing)

Decoding Machine Language

- What is the assembly language of the following machine instruction?

0x00af8020

- Hex to bin: 0000 0000 1010 1111 1000 0000 0010 0000

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	100000

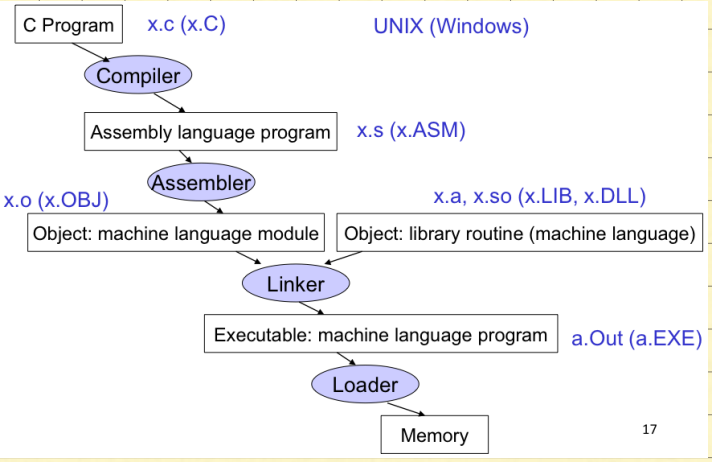
- Get the instruction: add $s0,$a1,$t7

op(31:26)								
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

Name	Fields						Comments
Field size	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits	All MIPS instructions are 32 bits long
R-format	op	rs	rt	rd	shamt	funct	Arithmetic instruction format
I-format	op	rs	rt	address/immediate			Transfer, branch, imm. format
J-format	op	target address					Jump instruction format

Starting a C Program



Role of Assembler

- Convert pseudo-instructions into actual hardware instructions – pseudo-instrs make it easier to program in assembly – examples: “move”, “blt”, 32-bit immediate operands, etc.

- Convert assembly instrs into machine instrs – a separate object file (x.o) is created for each C file (x.c) – compute the actual values for instruction labels – maintain info on external references and debugging information

将伪指令翻译成硬件指令.

将汇编指令翻译成机器指令.

Role of Linker

- Stitches different object files into a single executable
  - patch internal and external references
  - determine addresses of data and instruction labels
  - organize code and data modules in memory

- Some libraries (DLLs) are dynamically linked – the executable points to dummy routines – these dummy routines call the dynamic linker-loader so they can update the executable to jump to the correct routine

Object file 1:

Object file header			
	Name	Procedure A	
	Text size	100_{hex}	代码段
	Data size	20_{hex}	数据段
Text segment	Address	Instruction	
	0	lw $a0, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(X)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	lw	X 需指向X获得地址
	4	jal	B
Symbol table	Label	Address	
这些还未确定位置	X	–	
	B	–	

Object file 2:

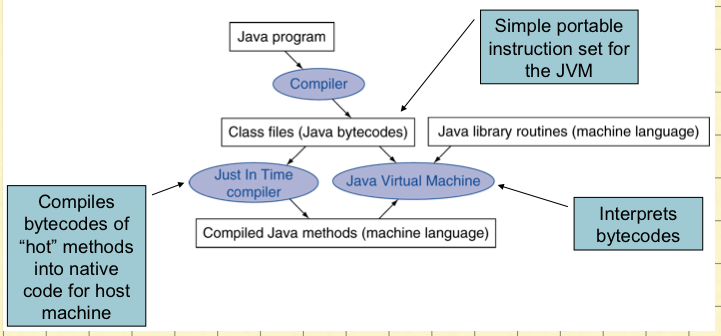
Object file header			
	Name	Procedure B	
	Text size	200_{hex}	
	Data size	30_{hex}	
Text segment	Address	Instruction	
	0	sw $a1, 0($gp)	
	4	jal 0	
	...	...	
Data segment	0	(Y)	
	...	...	
Relocation information	Address	Instruction type	Dependency
	0	sw	Y
	4	jal	A
Symbol table	Label	Address	
	Y	–	
	A	–	

Executable file: $\$gp+8000_{hex}=10008000_{hex}+ffff8000_{hex}=10000000_{hex}$

Executable file header		
	Text size	300_{hex} → file1 + file2
	Data size	50_{hex} → file1 + file2
Text segment	Address	Instruction
A	$0040\ 0000_{hex}$	lw $a0, 8000_{hex}($gp)
	$0040\ 0004_{hex}$	jal 40 0100_{hex}
	...	...
B	$0040\ 0100_{hex}$	sw $a1, 8020_{hex}($gp)
	$0040\ 0104_{hex}$	jal 40 0000_{hex}
	...	...
Data segment	Address	
X	$1000\ 0000_{hex}$	(X)
	...	...
Y	$1000\ 0020_{hex}$	(Y)
	...	...

根据相对位置决定

Starting Java Applications



The Swap Procedure

- Register allocation: $a0 and $a1 for the two arguments, $t0 for the temp variable – no need for saves and restores as we're not using $s0-$s7 and this is a leaf procedure (won't need to re-use $a0 and $a1)

```
swap:   sll   $t1, $a1, 2
        add   $t1, $a0, $t1
        lw    $t0, 0($t1)
        lw    $t2, 4($t1)
        sw    $t2, 0($t1)
        sw    $t0, 4($t1)
        jr    $ra
```

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

The Sort Procedure

- Register allocation: arguments v and n use $a0 and $a1, i and j use $s0 and $s1; must save $a0 and $a1 before calling the leaf procedure

- The outer for loop looks like this: (note the use of pseudo-instrs)

```
            move  $s0, $zero          # initialize the loop
loopbody1:  bge   $s0, $a1, exit1     # will eventually use slt and beq
            … body of inner loop …
            addi  $s0, $s0, 1
            j     loopbody1
exit1:
```

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

26

- The inner for loop looks like this:

```
            addi  $s1, $s0, -1        # initialize the loop
loopbody2:  blt   $s1, $zero, exit2   # will eventually use slt and beq
            sll   $t1,  $s1, 2
            add   $t2, $a0, $t1
            lw    $t3, 0($t2)
            lw    $t4, 4($t2)
            bgt   $t3, $t4, exit2
            … body of inner loop …
            addi  $s1, $s1, -1
            j     loopbody2
exit2:
```

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

27

Saves & Restores

- Since we repeatedly call "swap" with $a0 and $a1, we begin "sort" by copying its arguments into $s2 and $s3 – must update the rest of the code in "sort" to use $s2 and $s3 instead of $a0 and $a1

- Must save $ra at the start of "sort" because it will get over-written when we call "swap"

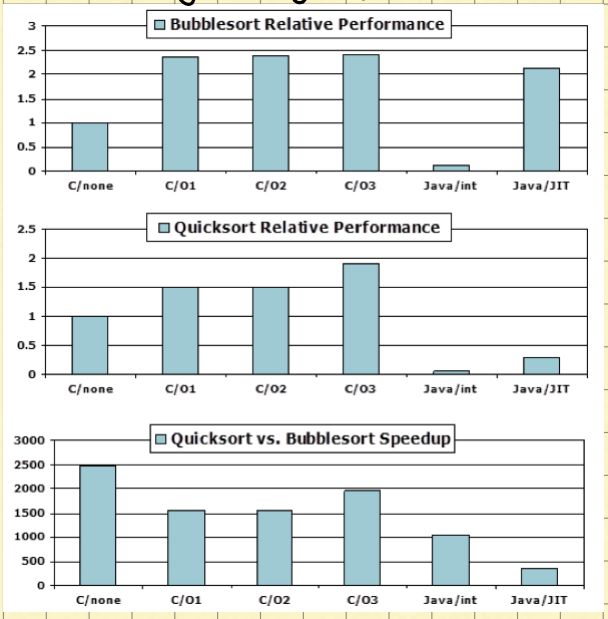
- Must also save $s0-$s3 so we don't overwrite something that belongs to the procedure that called "sort"

```
sort:   addi  $sp, $sp, -20
        sw    $ra, 16($sp)
        sw    $s3, 12($sp)
        sw    $s2, 8($sp)
        sw    $s1, 4($sp)
        sw    $s0, 0($sp)
        move  $s2, $a0
        move  $s3, $a1
          …
        move  $a0, $s2      # the inner loop body starts here
        move  $a1, $s1
        jal   swap
          …
exit1:  lw    $s0, 0($sp)
          …
        addi  $sp, $sp, 20
        jr    $ra
```

9 lines of C code → 35 lines of assembly

29

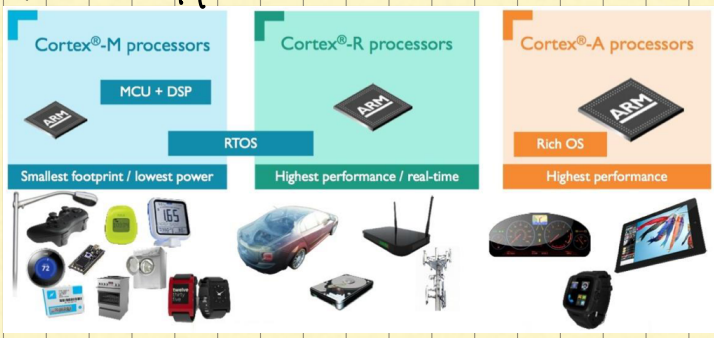
Effect of Language and Algorithm



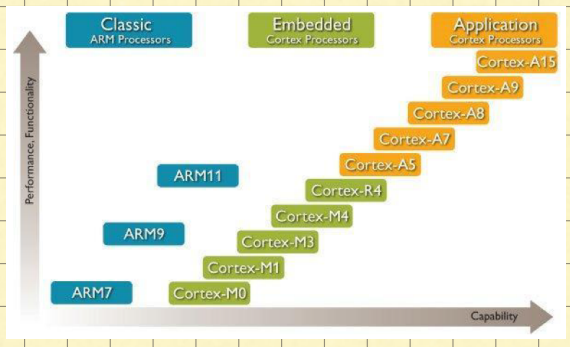
- Instruction count and CPI are not good performance indicators in isolation
- Compiler optimizations are sensitive to the algorithm
- Java/JIT compiled code is significantly faster than JVM interpreted
  - Comparable to optimized C in some cases
- Nothing can fix a dumb algorithm!

①指令数、CPI
②编译器
③算法

ARM Applications



ARM CPU Series



ARM & MIPS Similarities

- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

ARM v8 Instructions

- In moving to 64-bit, ARM did a complete overhaul
- ARM v8 resembles MIPS
  - Changes from v7:
    - No conditional execution field
    - Immediate field is 12-bit constant
    - Dropped load/store multiple
    - PC is no longer a GPR
    - GPR set expanded to 32
    - Addressing modes work for all word sizes
    - Divide instruction
    - Branch if equal/branch if not equal instructions

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
- 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
- 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