COMP 30080 Processor Design

2. MIPS Instructions: Language of the Computer

Dr Chris Bleakley

UCD School of Computer Science and Informatics.

Scoil na Ríomheolaíochta agus an Faisnéisíochta UCD.

Introduction

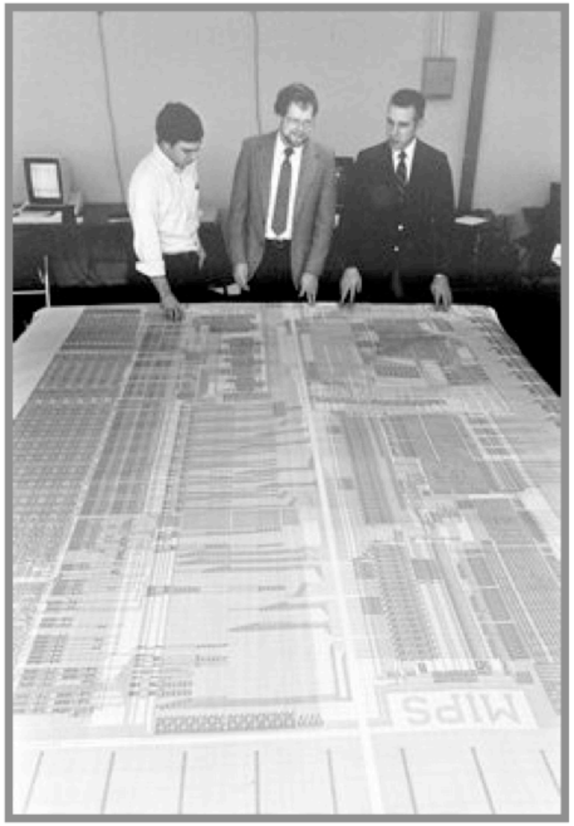
About MIPS

MIPS (Microprocessor without Interlocked Pipeline Stages)

- 1981 John Hennessy at Stanford started work on RISC (Reduced Instruction Set Computer) with deep pipeline.
- 1984 he set up MIPS Computer Systems
- 1985 created R2000, 32-bit RISC processor.
- 1991 created R4000, 1st 64-bit processor.
- 1992 acquired by main customer SGI (graphics workstations), became MIPS Technologies.
- 1990s licensed processors for embedded systems.
- 2012 MIPS Technologies acquired by Imagination Technologies

MIPS

The Heritage of the MIPS Architecture

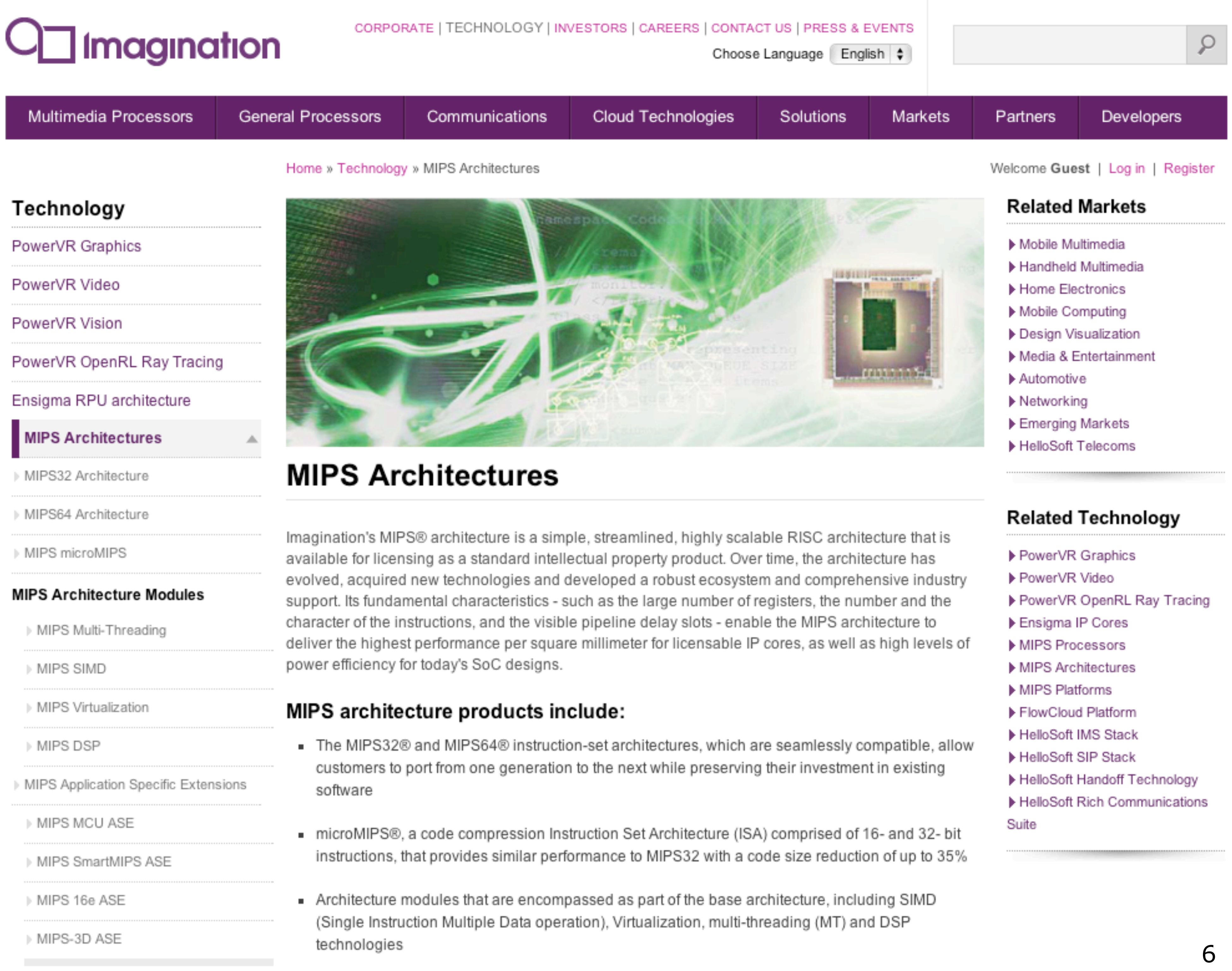


Pioneered by Stanford President John Hennessy in the 1980s

Pure, fast, efficient, elegant RISC architecture designed for performance

Now the architecture of choice for multimedia, home networking & beyond

Innovation continues by MIPS and licensees—Altera, Broadcom, Cavium, ICT, NEC, NetLogic, Toshiba, others



Home » Technology » MIPS Architectures

Welcome **Guest** | Log in | Register

Technology

- PowerVR Graphics
- PowerVR Video
- PowerVR Vision
- PowerVR OpenRL Ray Tracing
- Ensigma RPU architecture
- **MIPS Architectures**
  - MIPS32 Architecture
  - MIPS64 Architecture
  - MIPS microMIPS

MIPS Architecture Modules

- MIPS Multi-Threading
- MIPS SIMD
- MIPS Virtualization
- MIPS DSP
- MIPS Application Specific Extensions
  - MIPS MCU ASE
  - MIPS SmartMIPS ASE
  - MIPS 16e ASE
  - MIPS-3D ASE

MIPS Architectures

Imagination's MIPS® architecture is a simple, streamlined, highly scalable RISC architecture that is available for licensing as a standard intellectual property product. Over time, the architecture has evolved, acquired new technologies and developed a robust ecosystem and comprehensive industry support. Its fundamental characteristics - such as the large number of registers, the number and the character of the instructions, and the visible pipeline delay slots - enable the MIPS architecture to deliver the highest performance per square millimeter for licensable IP cores, as well as high levels of power efficiency for today's SoC designs.

MIPS architecture products include:

- The MIPS32® and MIPS64® instruction-set architectures, which are seamlessly compatible, allow customers to port from one generation to the next while preserving their investment in existing software
- microMIPS®, a code compression Instruction Set Architecture (ISA) comprised of 16- and 32- bit instructions, that provides similar performance to MIPS32 with a code size reduction of up to 35%
- Architecture modules that are encompassed as part of the base architecture, including SIMD (Single Instruction Multiple Data operation), Virtualization, multi-threading (MT) and DSP technologies

Related Markets

- Mobile Multimedia
- Handheld Multimedia
- Home Electronics
- Mobile Computing
- Design Visualization
- Media & Entertainment
- Automotive
- Networking
- Emerging Markets
- HelloSoft Telecoms

Related Technology

- PowerVR Graphics
- PowerVR Video
- PowerVR OpenRL Ray Tracing
- Ensigma IP Cores
- MIPS Processors
- MIPS Architectures
- MIPS Platforms
- FlowCloud Platform
- HelloSoft IMS Stack
- HelloSoft SIP Stack
- HelloSoft Handoff Technology
- HelloSoft Rich Communications Suite

MIPS

http://www.imgtec.com/mips

- "Embedded system is a special-purpose computer system, which is completely encapsulated by the device that it controls."
- Embedded processor companies license processor designs for inclusion in special-purpose chips.

MIPS

- 1 MIPS processor takes up <1.5 mm^2 of die area (0.18um process).
- Used in:
  - Digital TVs, Set-top boxes, Blu-ray players, DVD players, Digital cameras, WiFi access points, Printers, PlayStation2, PSP, smartphones, tablets
- Competitors:
  - PowerPC, ARM, Intel Quark (IA32)
- Processor landscape…

MIPS

- Why MIPS for this course?
  - 'Clean' ISA – logical, few special cases
  - Comparatively easy to understand
  - Good texts
  - Tool support
  - Principals learnt are applicable to other architectures
  - Even classic CISC architectures such as IA-32 now use RISC concepts internally in the processors

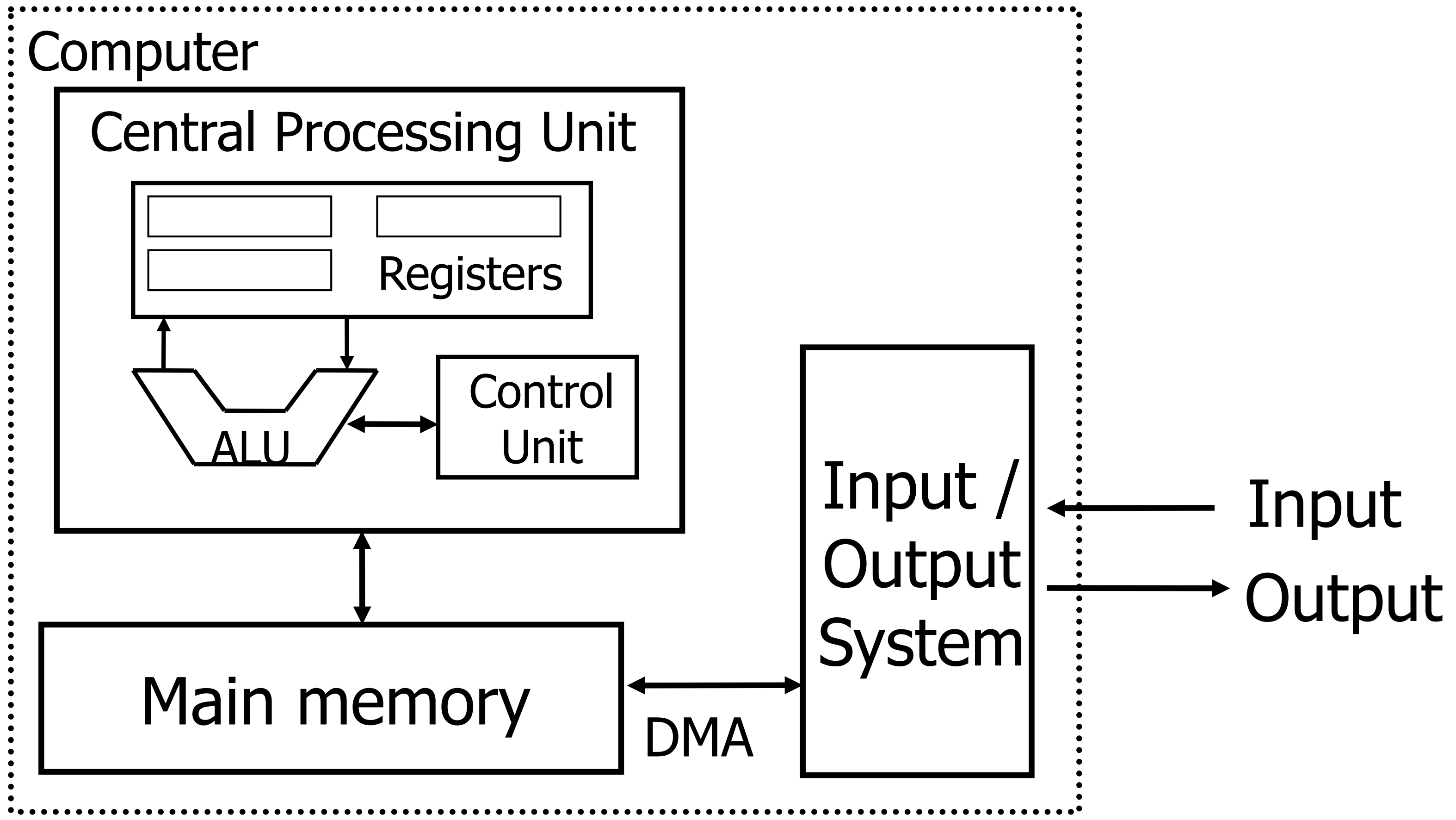
- Nowadays two ISA variants MIPS-32 and MIPS-64

$\uparrow$

this course

MIPS Assembly Language

MIP32 Architecture



32-bit words in memory
32 data registers, each storing a 32-bit word

Arithmetic Instructions

```
a = b + c;
```

C equivalent
3 variables

Variable	Register
a	$s0
b	$s1
c	$s2

Register map

MIPS instruction

```
add  $s0, $s1, $s2   # data in registers
```

operation

a b c

operands - register names

comment

MUST HAVE 3 AND ONLY 3 OPERANDS

Arithmetic Instructions

```
d = b - c;
```

Variable	**Register**
d	$s3
b	$s1
c	$s2

```
sub  $s3, $s1, $s2   # data in registers
```

Pop Quiz

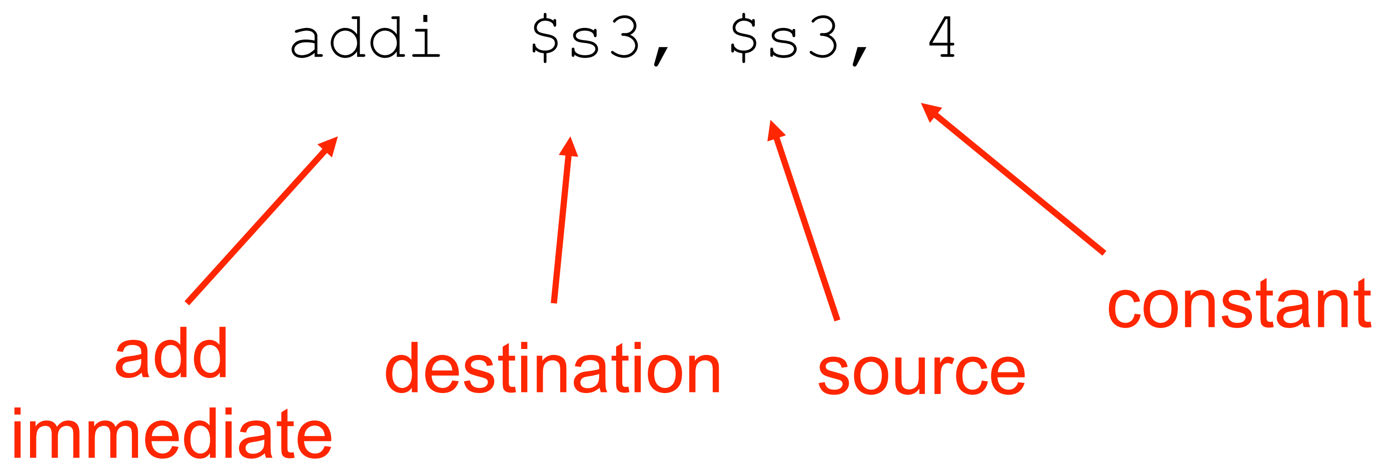
- How to deal with more than three variables?

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

Variable	Register
`f`	`$s0`
`g`	`$s1`
`h`	`$s2`
`i`	`$s3`
`j`	`$s4`

Arithmetic Instructions

```
a = a + 4;
```



```
addi  $s3, $s3, 4
```

Core Arithmetic Instructions

Instruction	**MIPS Example**	**C Equivalent**
Add	`add $t0,$t1,$t2`	`t0 = t1 + t2`
Add immediate	`addi $t0,$t1,2`	`t0 = t1 + 2`
Subtract	`sub $t0,$t1,$t2`	`t0 = t1 - t2`

Memory Instructions

element number 8 of array

```
g = h + a[2];
```

destination

offset of element (in **<u>bytes</u>**) relative to base

load word

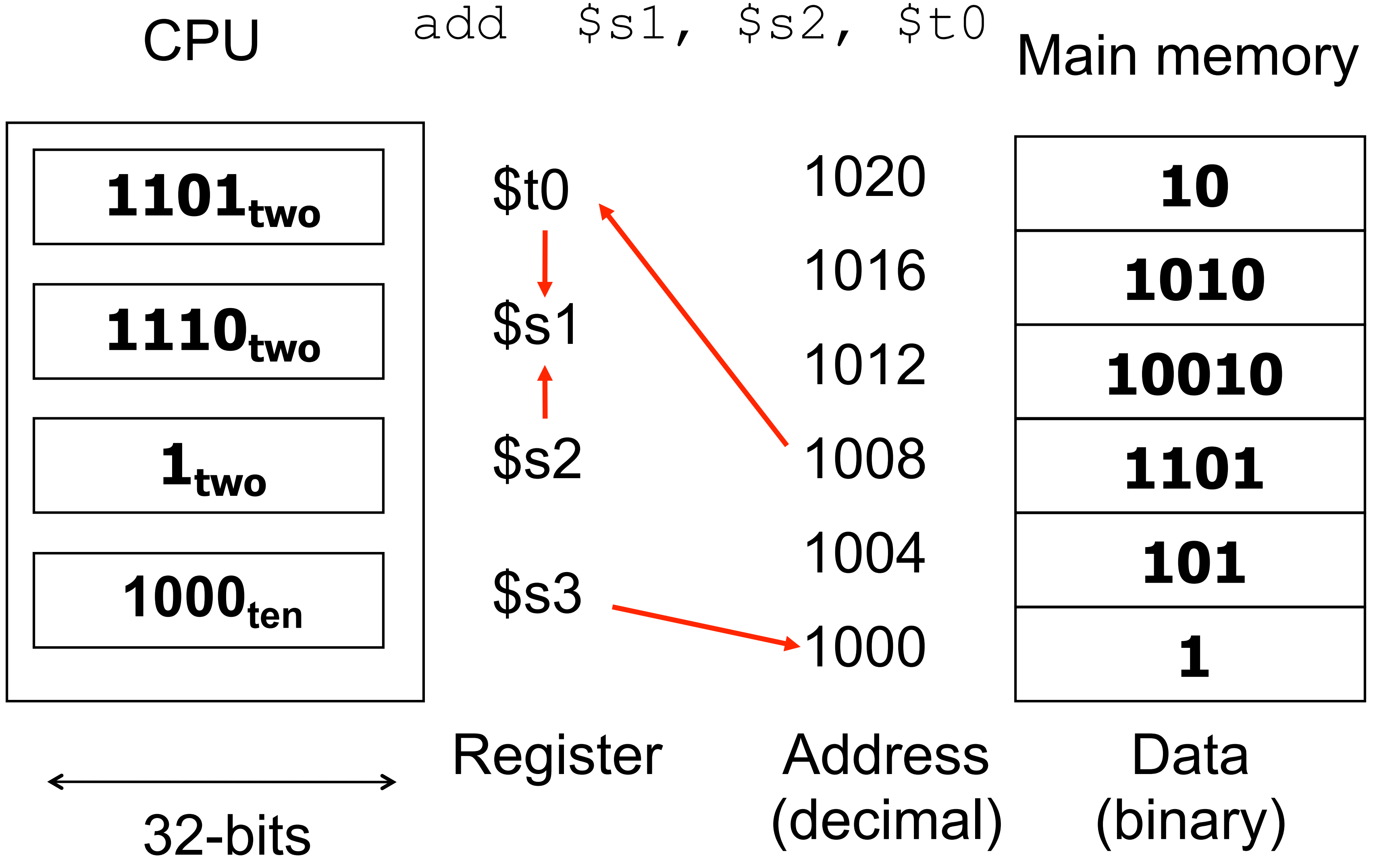
register containing base address of array a

```
lw    $t0, 8($s3)          # load a[2] to $t0
add   $s1, $s2, $t0        # g = h + a[2]
```

actual address = base + offset

Memory Instructions

```
lw    $t0, 8($s3)
add   $s1, $s2, $t0
```



Memory Instructions

- The bytes within a word can be addressed in one of two ways.

- Big endian (MIPS, Motorola)

Data	MSB			LSB
Byte number	0	1	2	3

- Little endian (Intel)

Data	MSB			LSB
Byte number	3	2	1	0

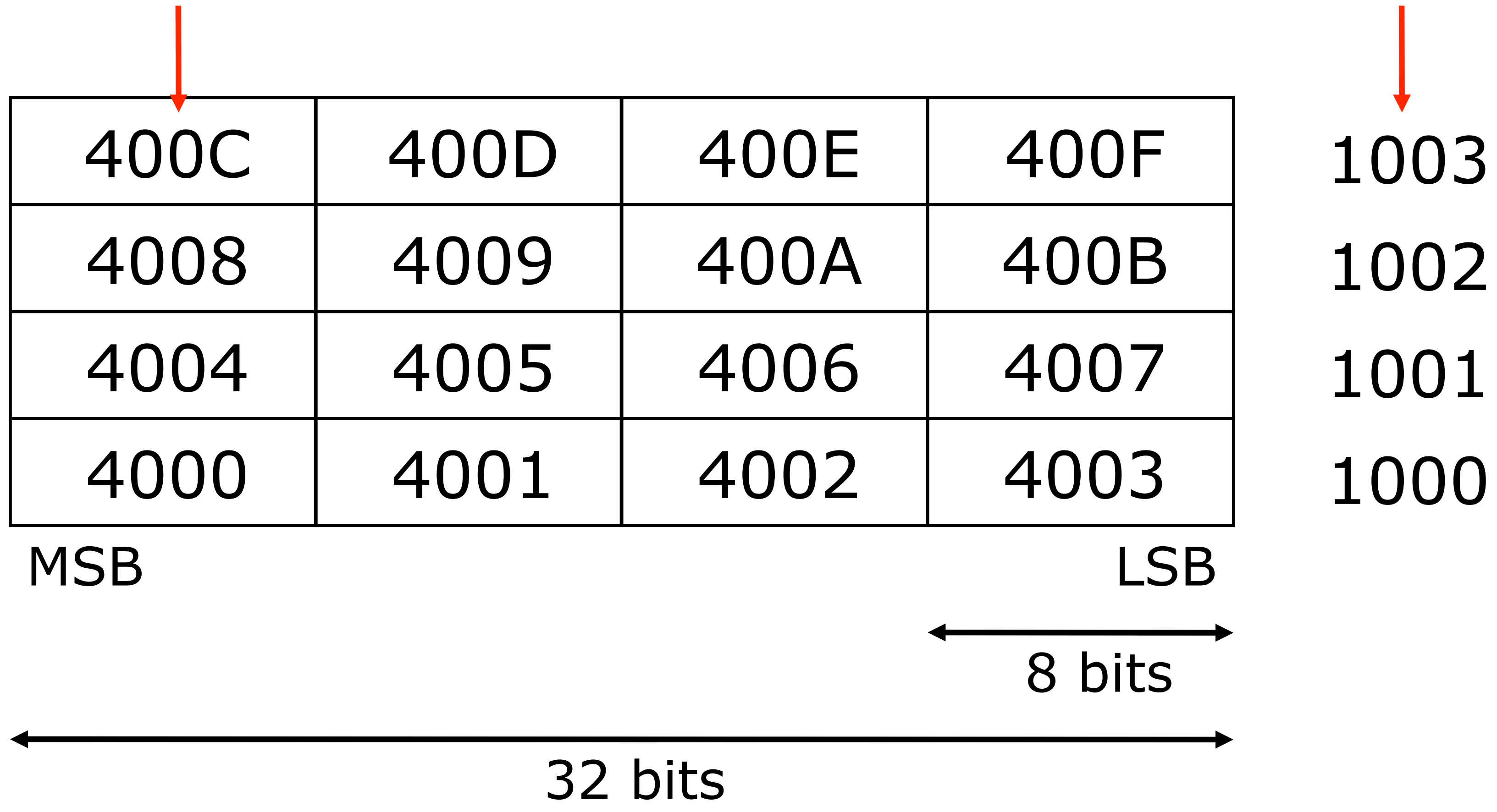
Memory Instructions

- Note, SPIM/MARS uses the format of the underlying computer. So, on Intel (PCs and new Macs) SPIM/MARS uses Little Endian.

Memory Instructions

Data access
Byte addressing
Allows access
to characters

Instruction access
Word addressing
Allows lots
of instructions



Memory Instructions

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

Variable	Register
h	$s2
*a[0]	$s3

Address of / pointer to

```
lw    $t0, 32($s3)      # load a[8]
add   $t0, $s2, $t0     # calculate
sw    $t0, 48($s3)      # store result
```

store word

source

destination

Pop Quiz

- Implement the following in MIPS32 assembly language

```
y = x[1]+2;
```

Variable	Register
y	$t0
*x[0]	$t1

First MIPS Program

```
            .data
DataIn:     .word       1,2
DataOut:    .word       0
            .text
            .globl      main
main:       la          $t0, DataIn         # t0 = *DataIn
            lw          $t1, 0($t0)         # t1 = DataIn[0]
            lw          $t2, 4($t0)         # t2 = DataIn[1]
            add         $t3, $t1, $t2       # t3 = t1 + t2
            la          $t0, DataOut        # t0 = *DataOut
            sw          $t3, 0($t0)         # store t2 at *DataOut
            li          $v0, 10             # system call for exit
            syscall
```

First MIPS Program

```
            .data
DataIn:     .word       1,2
DataOut:    .word       0
            .text
            .globl      main
main:       la          $t0, DataIn          # t0 = *DataIn
            lw          $t1, 0($t0)          # t1 = DataIn[0]
            lw          $t2, 4($t0)          # t2 = DataIn[1]
            add         $t3, $t1, $t2        # t3 = t1 + t2
            la          $t0, DataOut         # t0 = *DataOut
            sw          $t3, 0($t0)          # store t2 at *DataOut
            li          $v0, 10              # system call for exit
            syscall
```

Labels. Translate to memory addresses.

Start of data segment directive.

Stores the following data in main memory as words.

Start of text segment directive.

Global label

Load Address Pseudo-instruction.

System function call. In this case a return value of 10 is used to terminate the program.

Core Logical Instructions

Instruction	**MIPS Example**	**C Equivalent**
Bit-by-bit AND	`and  $s1,$s2,$s3`	`s1 = s2 & s3`
Bit-by-bit OR	`or   $s1,$s2,$s3`	`s1 = s2 | s3`
Bit-by-bit NOR	`nor  $s1,$s2,$s3`	`s1 = ~(s2 | s3)`
Bit-by-bit AND with constant	`andi $s1,$s2,100`	`s1 = s2 & 100`
Bit-by-bit OR with constant	`ori  $s1,$s2,100`	`s1 = s2 | 100`
Shift left logical by constant	`sll  $s1,$s2,10`	`s1 = s2 << 10`
Shift right logical by constant	`srl  $s1,$s2,10`	`s1 = s2 >> 10`

Pop Quiz

- Set a Boolean to indicate if variable *a* is odd (TRUE) or not (FALSE).

Variable	Register
`*a`	`$t0`

Pop Quiz

- Implement the following in MIPS32 assembly language:

```
y = x[i] + a;
```

Variable	**Register**
`*a`	`$t0`
`*x[0]`	`$t1`
`i`	`$s0`

Flow Control

```
goto L1;
…
L1: …
```

label

jump destination

VERY, VERY NAUGHTY!!

jump (unconditional)

```
        j   L1
        …
L1:     …
```

label, assembler puts in actual address of destination instruction

destination instruction

Flow Control

```
if (a == b)
  goto L2;
…
L2: …
```

label

jump destination

```
    beq  $s3, $s4, L2
    …
L2: …
```

compare

branch if equal,
otherwise continue

Flow Control

```
if (a != b)
    goto L3;
…
L3: …
```

```
      bne  $s3, $s4, L3      #
      …
L3:   …                      # branch destination
```

branch if not equal

Pop Quiz

```
if (a == b)
   f = g + h;
else
   f = g - h;
```

Variable	Register
f	$s0
g	$s1
h	$s2
a	$s3
b	$s4

Loops

- Consider the common types of loops:

```
for (i=0; i<N; i++) {body}

while (condition) {body}
```

pre-test

```
repeat {body} while (condition)
```

post-test

Loops

```
for (i=0; i<8, i++) {body}
```

```
      addi   $t0, $zero, 8
Loop: body                          # i not used in body
      addi   $t0, $t0, -1
      bne    $t0, $zero, Loop
```

OR

```
      addi   $t0, $zero, 0
      addi   $t1, $zero, 8
Loop: body                          # i used in body
      addi   $t0, $t0, +1
      bne    $t0, $t1, Loop
```

Loops

```
while (x == 3) {body}
```

```
     addi  $t1, $zero, 3
Loop:bne   $t0, $t1, Exit   # calculate condition
     body
     j      Loop
Exit:…
```

Loops

```
repeat {body} while (x == 3)
```

```
     addi   $t1, $zero, 3
Loop:body
     beq    $t0, $t1, Loop # calculate condition
```

Pop Quiz

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

Variable	Register
i	$s3
*b[0]	$s6
k	$s5

Core Branch and Jump Instructions

Instruction	MIPS Example	C Equivalent

Branch on equal	`beq $s1,$s2,Label`	`if (s1 == s2)`
		`  goto Label`

Branch on not equal	`bne $s1,$s2,Label`	`if (s1 != s2)`
		`  goto Label`

Jump	`j Label`	`goto Label`
Jump register	`jr $t0`	
Jump and link	`jal Label`	`Label`

- Handy feature:
$zero is a special register whose value is fixed to zero.

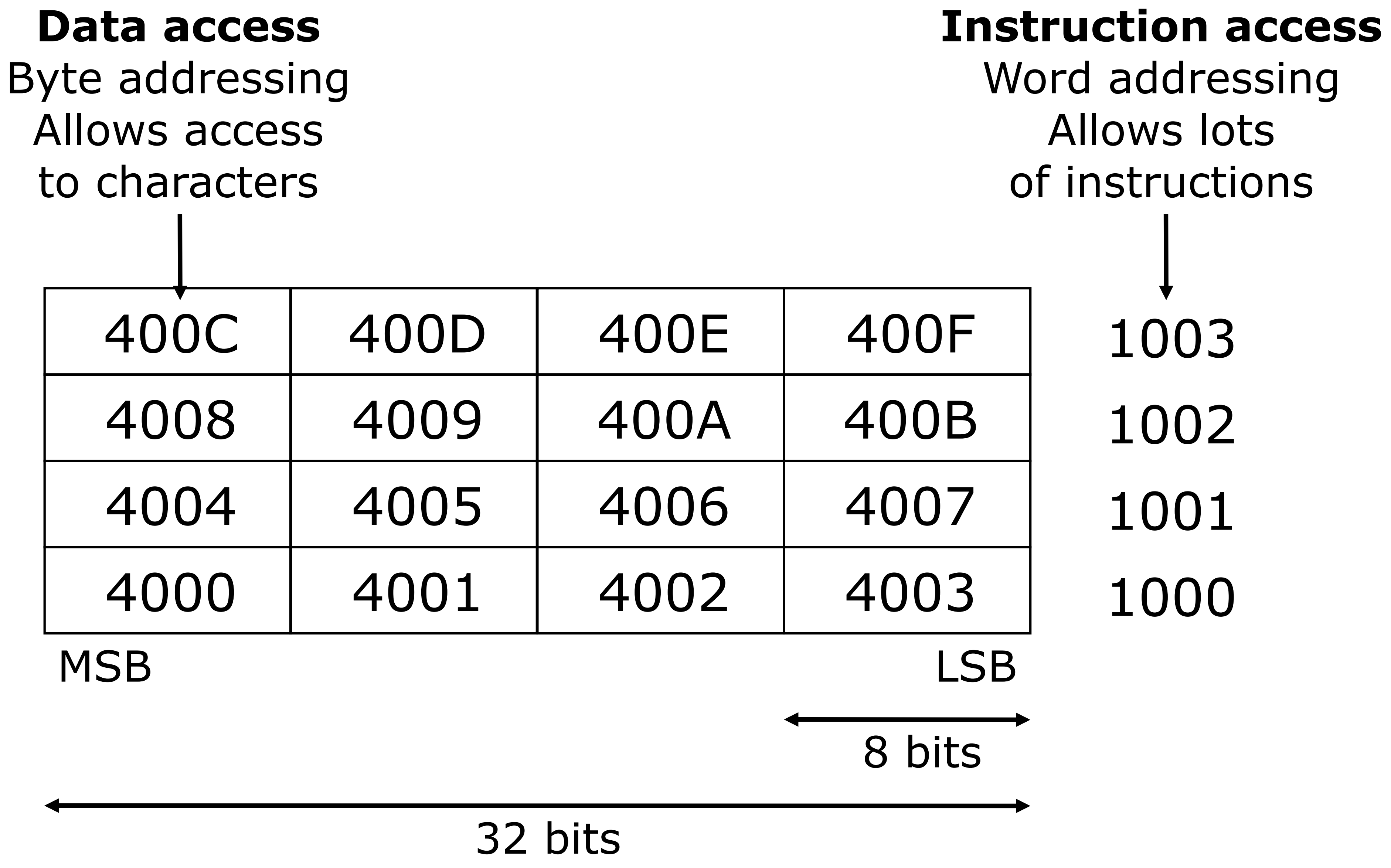
Set Instructions

Instruction	**MIPS Example**	**C Equivalent**

Set 1st operand (to 1) if 2nd operand is less than 3rd operand	`slt  $s1,$s2,$s3`	`if (s2 < s3)`
		`  s1 = 1;`
		`else`
		`  s1 = 0;`

Set 1st operand (to 1) if 2nd operand is less than constant	`slti $s1,$s2,10`	`if (s2 < 10)`
		`  s1 = 1;`
		`else`
		`  s1 = 0;`

Reminder…


Data access
Byte addressing
Allows access
to characters
Instruction access
Word addressing
Allows lots
of instructions
400C
400D
400E
400F
1003
4008
4009
400A
400B
1002
4004
4005
4006
4007
1001
4000
4001
4002
4003
1000
MSB
LSB
8 bits
32 bits

More Memory Instructions

- MIPS supports 8-bit (byte) load and stores. This is useful for text processing (ASCII characters). The rightmost bits of the register are used:

```
lb   $t0, 0($sp)    # copy 8-bit
sb  $t0, 0($gp)
```

- MIPS support 16-bit (half-word) load and stores:

```
lh   $t0, 0($sp)    # copy 16-
bits
sh  $t0, 0($gp)
```

Core Memory Instructions

Instruction	**MIPS Example**	**C Equivalent**
Load word	`lw $s1,4($t0)`	
Store word	`sw $s1,4($t0)`	
Load half-word	`lh $s1,4($t0)`	
Store half-word	`sh $s1,4($t0)`	
Load byte	`lb $s1,4($t0)`	
Store byte	`sb $s1,4($t0)`	

Procedures

- A procedure (function or routine) allows a programmer to encapsulate specific tasks. This aids clarity and reduces code size.
- There is a special instruction (jump and link) which simultaneously jumps to the subroutine and saves the current address plus one (PC+4) in the return address register ($ra):

```
jal ProcedureAddress
```

- Then the procedure is complete, a jump register instruction is used to load the return address into the program counter.

```
jr  $ra
```

Procedures

- MIPS defines special registers for procedure calls:
  - $a0 - $a3 : argument registers to pass parameters
  - $v0 - $v1 : value register to return results
  - $ra : return address
- The values in temporary registers need not be preserved by the called (callee) procedure:
  - $t0 - $t9 : temporary
- The values in saved registers must be preserved:
  - $s0 - $s7 : saved registers
- If the temporary registers are all used, extra data must be spilled to memory.
- The stack provides hardware support for spilling.

Procedures

- Stack is Last-In First-Out (LIFO) queue.
- Stack Pointer ($sp register) holds address of top of stack (last saved item). Stack grows top-down.

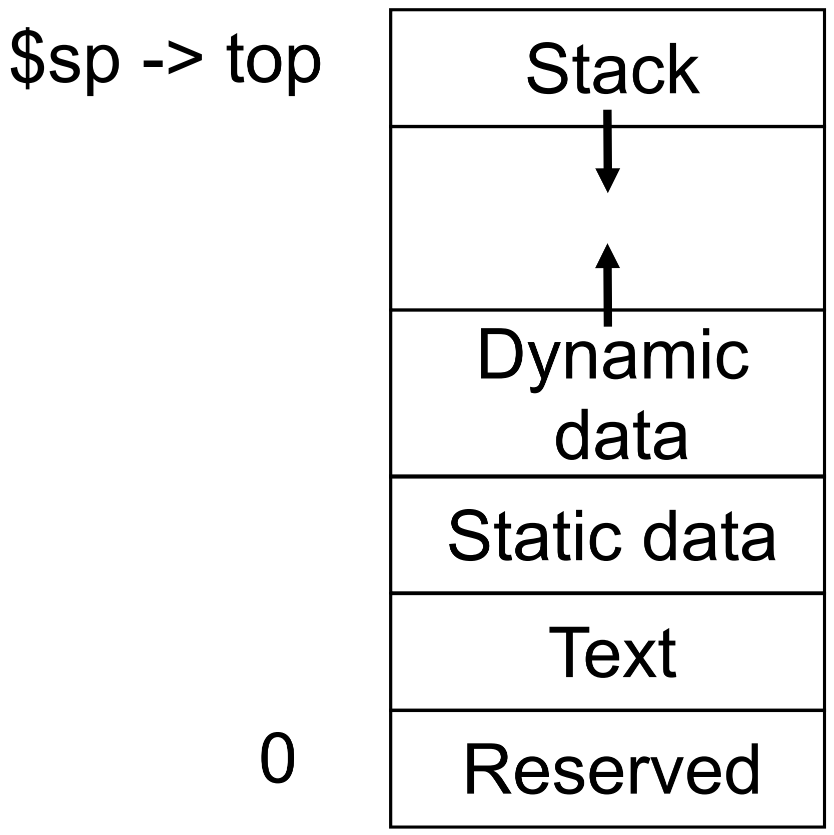
```
addi    $sp, $sp, -12          # move sp down
sw      $t1, 8($sp)            # push $t1
sw      $t0, 4($sp)            # push $t0
sw      $s0, 0($sp)            # push $s0
…                              # regs available
lw      $s0, 0($sp)            # pop $s0
lw      $t0, 4($sp)            # pop $t0
lw      $t1, 8($sp)            # pop $t1
addi    $sp, $sp, 12           # move sp back up
```

Procedures

- Note, the Stack Pointer and the Stack above the Stack Pointer must be preserved during a procedure call.
- When using a nested procedure call, the return address ($ra) must also be preserved. The simplest way is to push its contents to the stack.

Memory Model

- Text segment contains program machine instructions.
- Static data segment contains constants and other static variables.
- Dynamic data segment is used for dynamically variable data structures (the heap). Java new and C malloc.



Register List

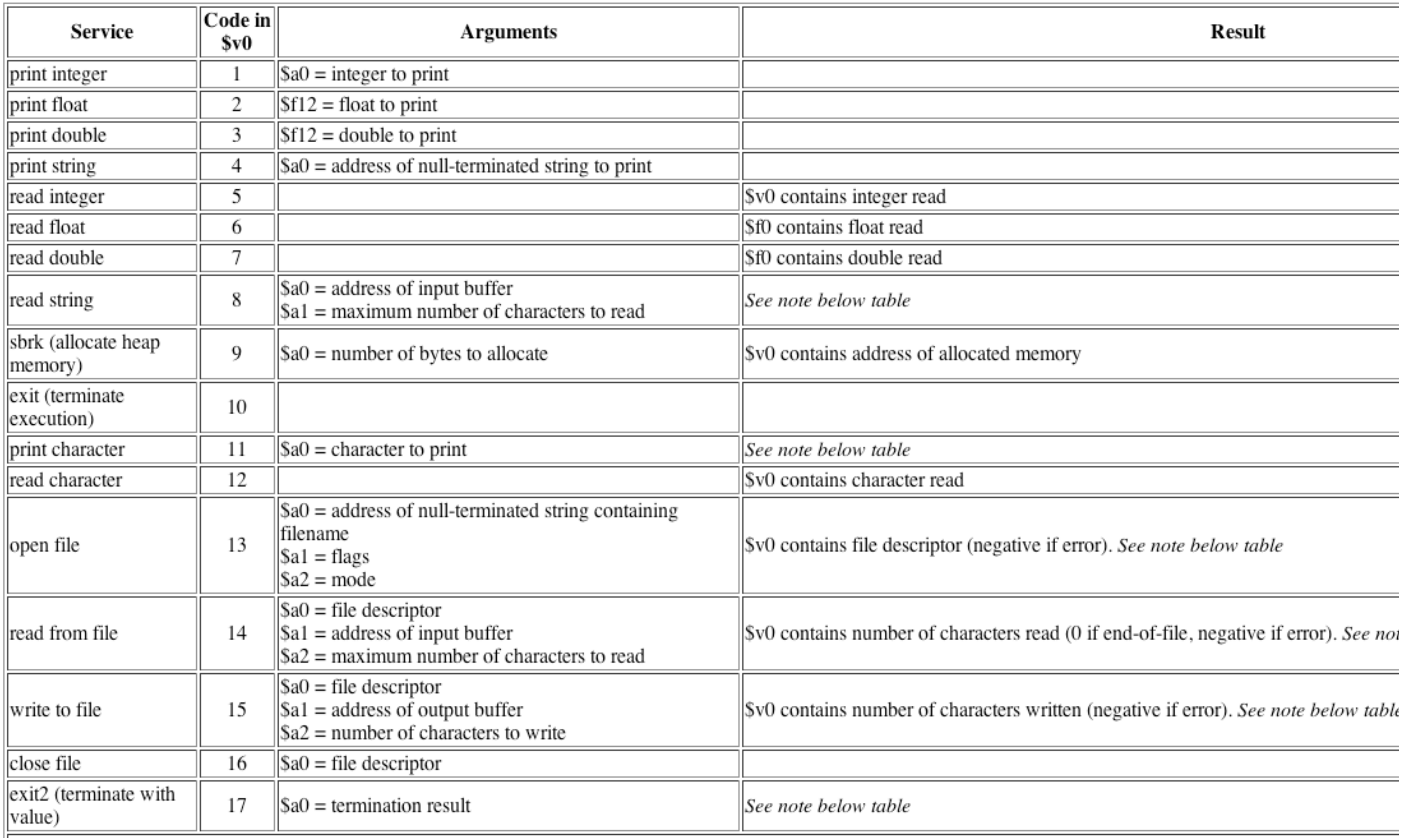
Name	Number	Usage	Preserved
$zero	0	Constant	n/a
$at	1	Assembler temporary	No
$v0 - $v1	2–3	Result values	No
$a0 - $a3	4–7	Arguments	No
$t0 - $t7	8–15	Temporary	No
$s0 - $s7	16–23	Saved	Yes
$t8 - $t9	24–25	More temporary	No
$k0-$k1	26–27	Reserved for OS kernel	n/a
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Syscall

- Library of function are provided with the MIPS simulator.
- To use one:
  1. Load the service number into $v0.
  2. Load the argument values (if any) into $a0, etc.
  3. Issue syscall instruction.
  4. Retrieve return value (if any).
- E.g.

```
li     $v0, 1                # service 1 is print integer
add    $a0, $t0, $zero       # load desired value
syscall
```

Syscall



Service	Code in $v0	Arguments	Result
print integer	1	$a0 = integer to print	
print float	2	$f12 = float to print	
print double	3	$f12 = double to print	
print string	4	$a0 = address of null-terminated string to print	
read integer	5		$v0 contains integer read
read float	6		$f0 contains float read
read double	7		$f0 contains double read
read string	8	$a0 = address of input buffer $a1 = maximum number of characters to read	*See note below table*
sbrk (allocate heap memory)	9	$a0 = number of bytes to allocate	$v0 contains address of allocated memory
exit (terminate execution)	10		
print character	11	$a0 = character to print	*See note below table*
read character	12		$v0 contains character read
open file	13	$a0 = address of null-terminated string containing filename $a1 = flags $a2 = mode	$v0 contains file descriptor (negative if error). *See note below table*
read from file	14	$a0 = file descriptor $a1 = address of input buffer $a2 = maximum number of characters to read	$v0 contains number of characters read (0 if end-of-file, negative if error). *See not*
write to file	15	$a0 = file descriptor $a1 = address of output buffer $a2 = number of characters to write	$v0 contains number of characters written (negative if error). *See note below tabl*
close file	16	$a0 = file descriptor	
exit2 (terminate with value)	17	$a0 = termination result	*See note below table*

Signed and Unsigned Numbers

- All the instructions up until here have assumed that the data is signed, i.e. 2s complement
- This means that:
  - Overflow of the results is handled with an exception
  - Sign extension is used when transferring data from fields to registers
- Some instructions have unsigned versions which do not do this…

Assembly Program Case Study

Case Study

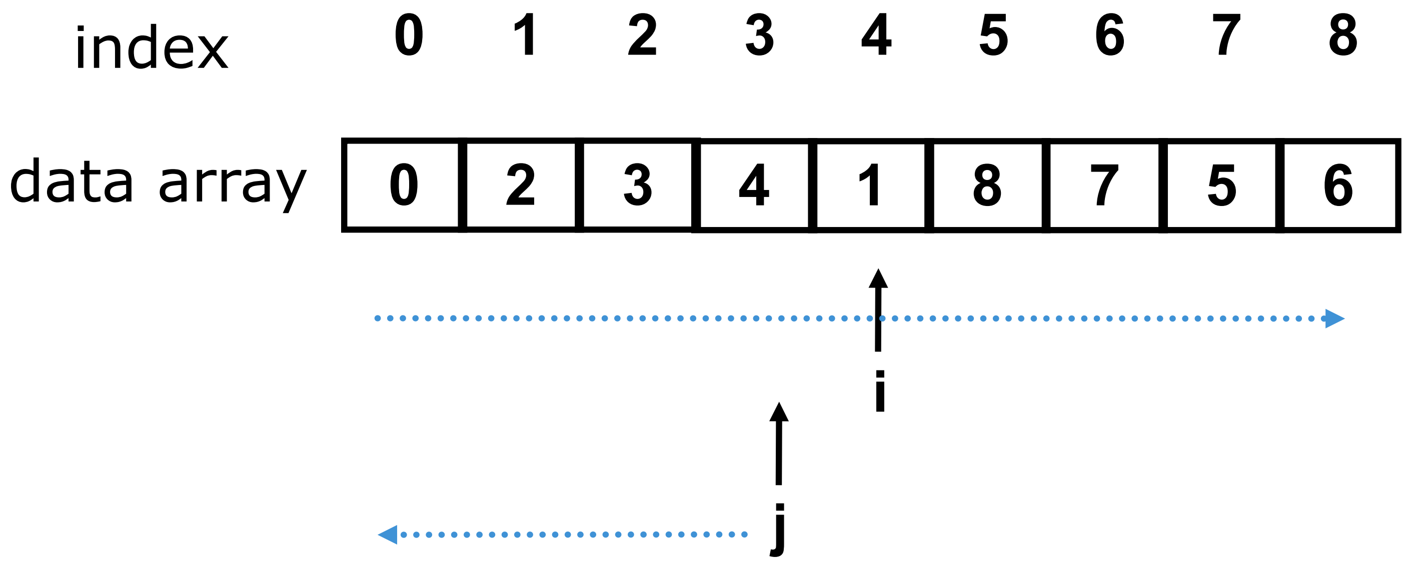
- Goal: write a program to sort a list of numbers.

1. Choose the algorithm
2. Write the program in C
3. Translate to assembly language

Case Study

- Algorithm – Bubble sort
  - Scan a pointer *i* from the beginning to the end of the list.
  - For each value of *i*, scan a second pointer *j* from the item immediately before *i* to the beginning of the list. If the value after *j* is less than the value at *j* then swap them.

Case Study



swap

Series of swaps moves 1 "bubble" to the correct position.
After that i is incremented and the process repeated.

Case Study

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

Case Study

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

Case Study

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

Variable	Register
*v[0]	$a0
k	$a1
temp	$t0

Pop Quiz

- Translate swap one line at a time

Case Study

```
swap: sll      $t1, $a1, 2     # $t1 = k*4
      add      $t1, $a0, $t1   # $t1 = *v[0]+k*4
      lw       $t0, 0($t1)     # $t0 = v[k]
      lw       $t2, 4($t1)     # $t2 = v[k+1]
      sw       $t2, 0($t1)     # v[k] = $t2
      sw       $t0, 4($t1)     # v[k+1] = $t0
      #
      jr   $ra
```

Variable	Register
*v[k]	$t1
k	$a1
v[k+1]	$t2

Case Study

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

Variable	Register
`*v[0]`	`$a0 & $s2`
`n`	`$a1 & $s3`
`i`	`$s0`
`j`	`$s1`

Case Study

- Outer loop:

```
for (i=0; i<n; i=i+1)
```

```
          move $s0, $zero          # i=0
forltst: slt  $t0, $s0, $s3        # set $t0 if i<n
          beq  $t0, $zero, exit1   # if $t0==0 then exit
             …
          addi $s0, $s0, 1         # i=i+1
          j    forltst             # jump loop start
exit1:    …
```

Case Study

- Inner loop

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

          addi $s1, $s0, -1         # j=i-1
for2tst: slti $t0, $s1, 0          # set $t0 if j<0
          bne  $t0, $zero, exit2    # if $t0==0 then exit
          sll  $t1, $s1, 2          # $t1 = j*4
          add  $t2, $s2, $t1        # $t2 = *v[0] + j*4
          lw   $t3, 0($t2)          # $t3 = v[j]
          lw   $t4, 4($t2)          # $t4 = v[j+1]
          slt  $t0, $t4, $t3        # set $t0 if v[j+1]<v[j
          beq  $t0, $zero, exit2    # if $t0==0 then exit
          …
          addi $s1, $s1, -1         # j=j-1
          j for2tst                 # jump to start loop
exit2:    …
```

Case Study

- Calling swap

```
move $s2, $a0            # copy *v[0]
move $s3, $a1            # copy n
…
move $a0, $s2            # $a0 = *v[0]
move $a1, $s1            # $a1 = j
jal  swap
```

Case Study

- Preserve registers on the stack

```
sort: addi      $sp, $sp, -20   # make room
      sw        $ra, 16($sp)    # push to stack
      sw        $s3, 12($sp)
      sw        $s2, 8($sp)
      sw        $s1, 4($sp)
      sw        $s0, 0($sp)
      …
      lw        $s0, 0($sp)     # pop from stack
      lw        $s1, 4($sp)
      lw        $s2, 8($sp)
      lw        $s3, 12($sp)
      lw        $ra, 16($sp)
      addi      $sp, $sp, 20     # restore stack
```

Case Study

- Test program that calls the sort procedure.

```
        .data
list:   .word  5,2,3,4,1,8,7,9,6
length: .word  9
        #
        .text
        .globl main

        #
        # Main procedure
        #

main:   addi $sp, $sp, -4       # store registers that will be messed up on the stack
        sw   $ra, 0($sp)
        la   $a0, list          # load address of list of numbers to argument 0
        la   $t0, length        # load address of length of list
        lw   $a1, 0($t0)        # load length of list to argument 1
        jal  sort               # jump to sort procedure
        lw   $ra, 0($sp)        # retore registers from stack
        addi $sp, $sp, 4
        jr   $ra                # return to loader
```

Case Study

- Complete program on Moodle

MIPS Machine Language

Representing Instructions

- Assembly language -> Machine language
- "The instruction format defines how the information in the assembly language instruction is coded into a binary machine language word."
- A 32-bit word is divided into a number of segments or fields. Each field is used to represent part of the instruction.
- To ensure that all instructions are 32-bits, MIPS designers chose to support a number of different instruction formats. The formats are distinguished using the opcode field.

Representing Instructions

- R-format (register format)

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

- op = operation (opcode), e.g. add
- rs = 1st register source operand (5 bits select from 32 regs)
- rt = 2nd register source operand
- rd = register destination operand
- shamt = shift amount
- funct = function, selects variant of operation in opcode field

Representing Instructions

- E.g. `add  $t0, $s1, $s2`

000000	10001	10010	01000	00000	100000
add	$s1	$s2	$t0	0	add

Representing Instructions

- I-format (immediate format)

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

- op = operation (opcode), e.g. lw, sw
- rs = base register
- rt = data register
- constant or address

Representing Instructions

Instruction syntax	Format	op	rs	rt	rd	shamt	funct	const/ address
`add` *rd*,*rs*,*rt*	R	0	reg	reg	reg	0	32_{ten}	-
`sub` *rd*,*rs*,*rt*	R	0	reg	reg	reg	0	34_{ten}	-
`addi` *rt*,*rs*,*c*	I	8_{ten}	reg	reg	-	-	-	constant
`lw` *rt*,*c*(*rs*)	I	35_{ten}	reg	reg	-	-	-	address
`sw` *rt*,*c*(*rs*)	I	43_{ten}	reg	reg	-	-	-	address

- R-format (register format)

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

- I-format (immediate format)

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

\$t0 - \$t7	8-15
\$s0 - \$s7	16-23

Pop Quiz

- Manually assemble

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

Variable	Register
h	$s2
*a[0]	$s3

```
lw    $t0, 32($s3)          # load a[8]
add   $t0, $s2, $t0         # calculate
sw    $t0, 48($s3)          # stores result
```

Reference

Instruction syntax	Format	op	rs	rt	rd	shamt	funct	const/ address
add *rd,rs,rt*	R	0	reg	reg	reg	0	32_{ten}	-
sub *rd,rs,rt*	R	0	reg	reg	reg	0	34_{ten}	-
addi *rt,rs,c*	I	8_{ten}	reg	reg	-	-	-	constant
lw *rt,c(rs)*	I	35_{ten}	reg	reg	-	-	-	address
sw *rt,c(rs)*	I	43_{ten}	reg	reg	-	-	-	address

R format

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

I format

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

Name	Number	Usage	Preserved
$zero	0	Constant	n.a.
$v0 - $v1	2-3	Result values	No
$a0 - $a3	4-7	Arguments	No
$t0 - $t7	8-15	Temporary	No
$s0 - $s7	16-23	Saved	Yes
$t8 - $t9	24-25	More temporary	No
$gp	28	Global pointer	Yes
$sp	29	Stack pointer	Yes
$fp	30	Frame pointer	Yes
$ra	31	Return address	Yes

Representing Instructions

- Conditional branch instructions & set immediate instructions use I-format
- Jump instruction uses J-format

op	address
6 bits	26 bits

Representing Instructions

- What if the address takes up more than 26 bits?

- There is a jump register instruction.
- R type.
- Allows 32-bit jumps

```
jr $s0
```

Representing Instructions

- How can you get a constant value into a register?

- For 16 bit constants:
  - Use ori with $zero (note addi sign extends)

```
ori   $s0, $zero, 24
```

- For 32 bit constants:

```
lui   $s0, 7          # load upper immediate
ori   $s0, $s0, 12
```

Representing Instructions

- What about branches? They are I-format, so only a 16-bit address is allowed.

- Use relative addressing.
- Relative to the address of the next instruction.

- Remember instructions use **word addresses.**

Representing Instructions

Byte address

Word address relative to **Exit** branch

```
[0x00400000] Loop: sll  $t1, $s3, 2
[0x00400004]       add  $t1, $t1, $s6          # -3
[0x00400008]       lw   $t0, 0($t1)            # -2
[0x0040000c]       bne  $t0, $s5, Exit         # -1
[0x00400010]       add  $s3, $s3, 1            # 0
[0x00400014]       j    Loop                   # +1
[0x00400018] Exit: …                           # +2
```

Label	**Byte Address**	**Relative Address**
Loop	0x0040_000	0xFFFA (–6)
Exit	0x0040_001	0x0002 (+2)

Pop Quiz

- What do you do if you need a conditional branch to an address displaced by 20 bits relative to the Program Counter?

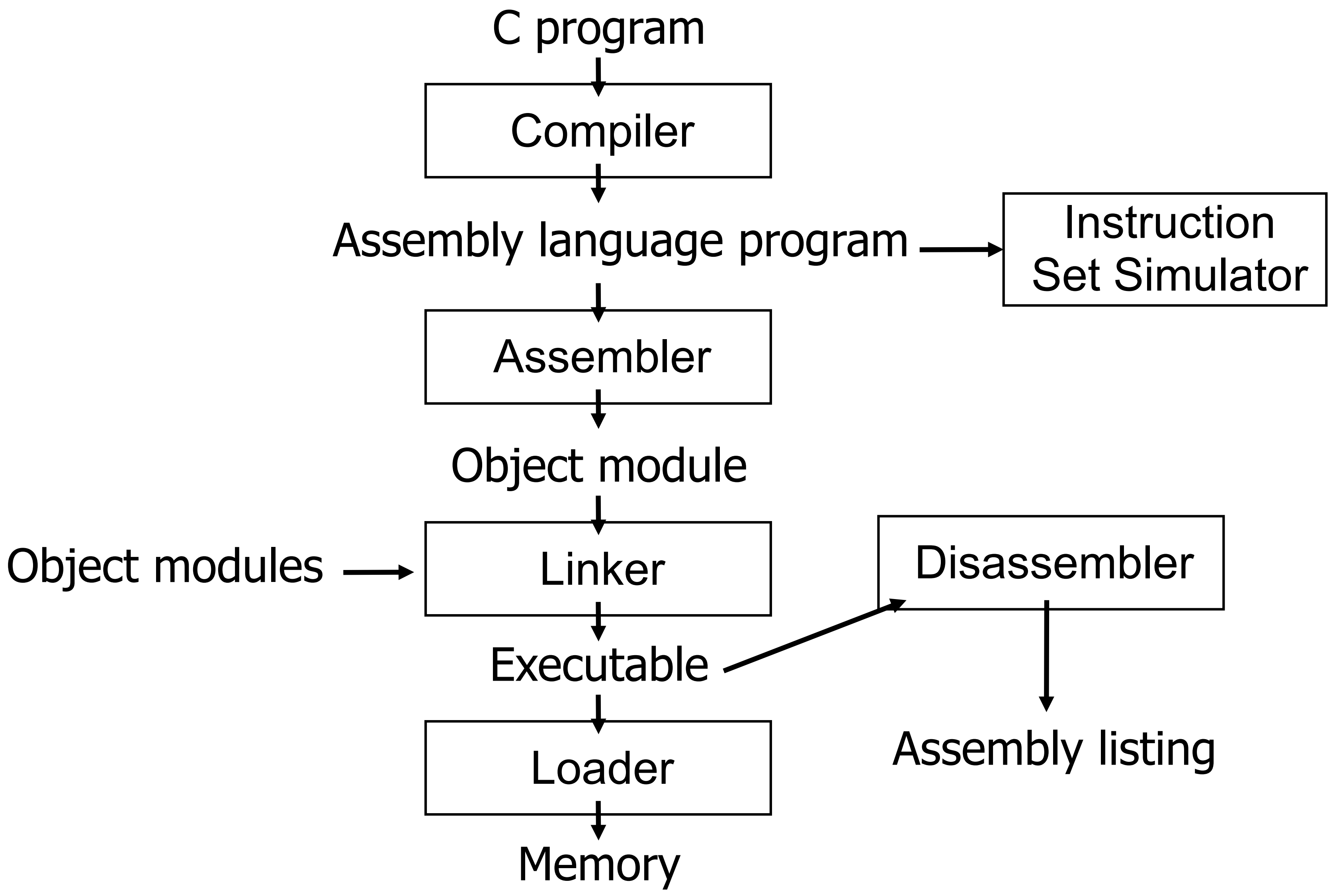
```
beq   $s0, $s1, L1    # BUT L1 is too far away!
```

Review of Addressing Modes

- Register addressing: “A variable’s value is stored in a register”. Used in add and sub.
- Immediate addressing: “A constant value is stored as part of the instruction”. Used in addi.
- Absolute addressing. "A variable or routine’s actual address in memory." Used by jumps.
- Relative addressing. "A variable or routine's address in memory is calculated with respect to the address of a specified instruction, typically the address of the next sequential instruction." Used by branches.
- Base or displacement addressing: “A variable or routine’s address is equal to the sum of the value in a register and a constant.” Use by load and store.

MIPS Tool Flow

Tool Flow


C program
Compiler
Assembly language program
Instruction Set Simulator
Assembler
Object module
Object modules
Linker
Disassembler
Executable
Loader
Assembly listing
Memory

Tool Flow
Compiler

- Transforms high-level program into assembly language program.

- >> see other courses !

Tool Flow
Assembler

- Converts assembly instructions to machine instructions.

1. Read instruction.
2. Check syntax. Ignore comments.
3. Put any labels or symbols in the symbol table.
4. Look up the instruction format.
5. Look up the instruction and register codes.
6. Write the machine instruction.
7. Once all instructions have been assembled, labels and symbols are resolved and actual values are put into the machine code program.

Tool Flow
Assembler

- “A pseudo instruction is a commonly used instruction that is not in the target instruction set but which the compiler accepts and translates to the target ISA equivalent”, e.g.:

```
move   $t0, $t1                # pseudo-instruction
add    $t0, $zero, $t1         # equivalent

li     $t0, 10                 # pseudo-instruction
ori    $t0, $zero, 10          # equivalent
```

Tool Flow
Assembler

- More pseudo instructions:

```
blt  $t0, $t1, L1 # branch less than
bgt  $t0, $t1, L1 # branch greater than
ble  $t0, $t1, L1 # branch less than or equal
bge  $t0, $t1, L1 # branch greater than or
equal

la   $s0, Label   # load address
li    $s0, 2       # load immediate
```

Tool Flow
Assembler

- “A directive tells the assembler how to translate the program but does not produce machine instructions”, e.g.:

```
.byte  20,12,30,12  # 8-bit data
.word  10,12        # 32-bit data
.ascii "ABC\n"      # character data
.align 2            # align to 2^2 byte boundary
.data               # start of data segment
.text               # start of program segment
.globl symbol       # declare symbol as global
                    # allows external reference
.space 40           # allocates 40 bytes of
                    # memory space
                    # in data segment only
```

Tool Flow
Assembler

- Object module files:
  - NOTHING TO DO WITH OBJECT-ORIENTED PROGRAMMING.
  - Contain machine instructions, data and information to enable linking with other object modules to create a complete program.
  - Each object module typically contains a group of related procedures.
  - are not executable
  - can be relocated, i.e. they can be put anywhere in memory. To allow for this, a list is maintained of instructions and data words which need to be updated with absolute addresses prior to execution.
  - can call subroutines in other assembly programs. To allow for this, a list is maintained of instructions which need to be updated with the address of external labels prior to execution.

Assembler

- An object file contains 6 sections:
  - Header: gives size & position of other sections
  - Text segment: contains machine language program. May include unresolved external references.
  - Data segment: contains binary data. May include unresolved external references.
  - Relocation information: identifies instructions and data that depend on absolute addresses.
  - Symbol table: lists addresses with external labels and unresolved references.
  - Debugging information: relates source code to machine code to provide programmer with debug info.

Linker

- Combines independently assembled object modules to create an executable file.

1. Determines were code and data modules are placed in memory.
2. Determines the addresses of all data and instruction labels.
3. Updates all address references.

- Jargon: The link resolves all internal and external undefined labels and patches their references.

Tool Flow
Loader

- Puts an executable program into memory and starts it running.

1. Reads the file header to determine the size of the program and data.
2. Creates address space for the program in memory.
3. Copies the instructions and data from the executable file to the address space.
4. Copies program arguments to the stack.
5. Initialize the registers.
6. Jumps to a start-up routine that copies the arguments from the stack and calls the program's main routine.
7. When the main routine returns, the start-up routine terminates the program with an exit system call.

Tool Flow
Disassembler

- Converts machine instructions to assembly instructions.

1. [Manual only:] Convert hex to binary.
2. Inspect left-most 6 bits to determine opcode.
3. Use this to determine the fields in the rest of the instruction.
4. Look up the register codes to determine the register names.
5. When all the instructions have been decoded, determine the destinations of any jump, branch or call instructions. Add labels for clarity.

Disassembler

[31] [0]

$\boxed{\text{0000 00}}\text{00 1010 1111 1000 0000 0010 0000}$

R-type

op	rs	rt	rd	shamt	funct
000000	00101	01111	10000	00000	$\boxed{100000}$

add

```
add   $s0,$a1,$t7
```

Tool Flow
Instruction Set Simulator

- Takes assembly program as input
- Represents the current state of the processor by holding representation of memory and RAM contents in data structures.
- Reads each instruction and updates contents of data structures based on what the instruction would do.

Pop Quiz

- What happens if a program (in error) jumps to an address containing data?

MIPS Design Decisions

Design Decisions

- RISC
  - Meaning:
    - Reduced Instructions Set Computer
  - Why?
    - Allows for simple (and fast) interconnection of main memory, registers, ALU.
  - Consequences:
    - Several RISC instructions are needed to perform the same operations as some CISC instructions.
    - Simplifies compiler.
    - Relies on pipelining and high clock rate to achieve performance.

Design Decisions

- 32-bit architecture
  - Meaning:
    - Memory words and registers are 32-bits long.
  - Why?
    - Trade-off between speed of instruction and work done by that instruction.
    - Driven by practicalities of current technology and needs to current applications.
  - Consequences:
    - Instruction words must be 32-bits long.
    - Memory addresses must be 32-bits long, Therefore can only address 2^32 locations (bytes). Therefore maximum of 4 GB of main memory.

Design decisions

- 32 registers
  - Meaning:
    - CPU can store a maximum of 32 x 32-bit data words at any one time.
  - Why?
    - Trade-off between speed of accessing a register (more choices mean more logic gates) and causing delays by having to swap data in and out of main memory when the registers are full.
    - "Spilling is the process of putting less commonly used variables into main memory."
  - Consequences:
    - Indexing a particular register in a instruction word requires 5-bits (2^5 = 32 possibilities).

Design Decision

- All machine instructions fit in one memory word
  - Meaning:
    - Machine instructions are 32-bits long
  - Why?
    - Only a single fetch from memory is required in order to load an instruction.
  - Consequences:
    - Less instructions available, i.e. RISC.

Design decisions

- ALU only directly connected to registers (not to main memory)
  - Meaning:
    - Single instructions cannot mix memory accesses and arithmetic or logical operations.
  - Why?
    - It is way faster. The registers and the ALU can be closely interconnected. There is no delay waiting around to get something out of main memory.
  - Consequences:
    - Need separate instructions for accessing memory and using the ALU.

References

References

- Required reading
  - Patterson & Hennessy
    - Chapter 2, Appendix A (on CD-ROM)
  - Harris & Harris
    - Chapter 6
- To probe further
  - MIPS Technologies
  - MIPS32 Architecture for Programmers, volumes 1&2
  - Wikipedia
    - MIPS, SGI, Jim Clark, RISC, CISC, ARM, IA-32, Motorola 68K, PowerPC