MIPS Instruction Set Architecture (ISA)

Lecture 1
August 21st, 2017

Jae W. Lee (jaewlee@snu.ac.kr)
Computer Science and Engineering
Seoul National University

Download this lecture slides at https://goo.gl/rJPMQU

Slide credits: [CS:APP3e] slides from CMU; [COD5e] slides from Elsevier Inc.

Overview

Reference: [COD5e] Ch. 2

- **What is Instruction Set Architecture (ISA)?**
- **Operations of Computer Hardware**
- **Operands of Computer Hardware**
- **Logical Operations**
- **Instructions for Making Decisions**
- **Communicating with People (Handling Characters)**

Instruction Set Architecture*

**** Some slides are provided by Prof. Jinsoo Kim (SKKU)***

ISA Analogy – Interface vs. Implementation

- **Separation of interface from implementation**
  - Example: printf() function call in C

```
// In your code helloworld.c:
#include <stdio.h>
int main() {  printf ("Hello World!\n");  }
```

```
// In /usr/include/stdio.h:
// This file defines the interface of printf()
extern int printf (__const char *__restrict __format, ...);
```

```
// In ../glibc/stdio-common/printf.c:
// This file contains the actual implementation of printf()
int __printf (const char *format, ...)
{ ... }
```

ISA Analogy – Interface vs. Implementation

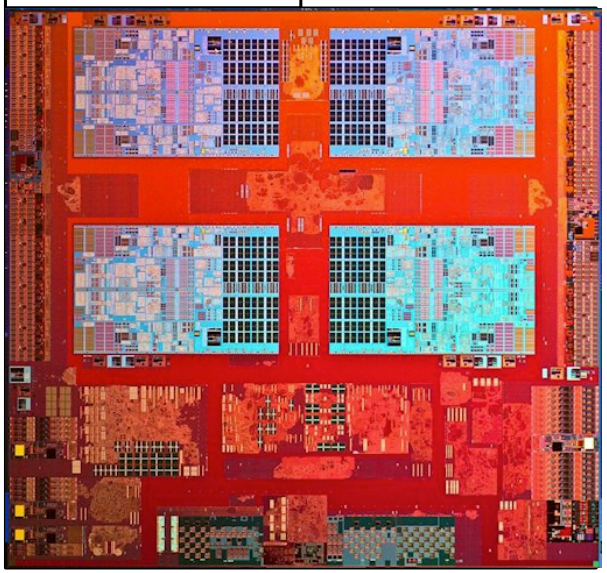
- **What if we apply the same idea to CPU?**
  - Separating interface (for users) from implementation (for designers)
  - Interface -> **architecture** / implementation -> **microarchitecture**

cpu.h

```
instruction set
instruction encoding
available registers
addressing modes
memory model
...
```

Architecture

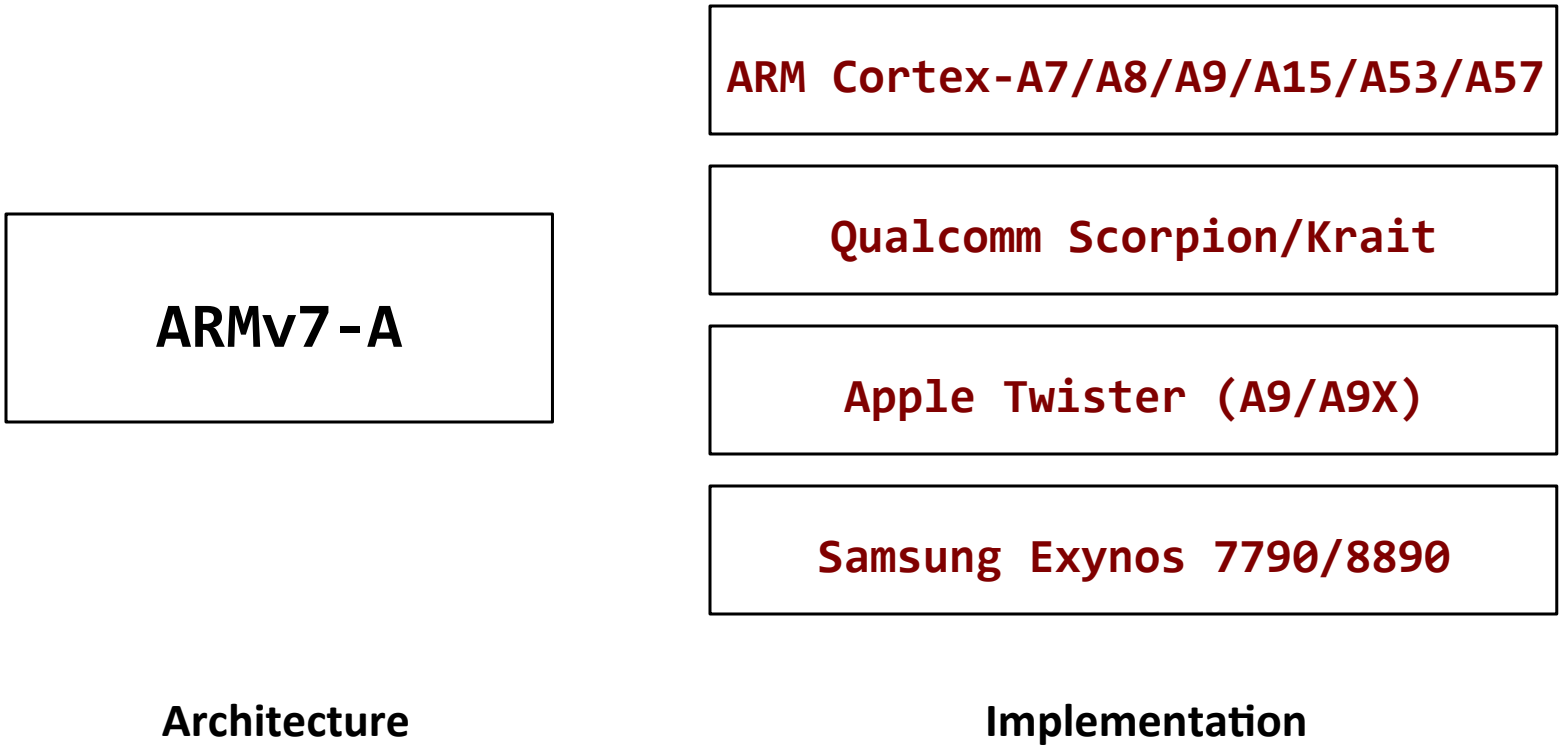
cpu.c



Implementation

ISA Analogy – Interface vs. Implementation

- **There may be many microarchitectures implementing the same architecture.**
  - Possibly from multiple vendors



ISA Analogy – Interface vs. Implementation

- **Examples: Architecture or Implementation?**
  - Number of software-visible general-purpose registers
  - Width of memory bus
  - Binary representation of the instruction: `sub r4,r2,#27`
  - Number of cycles to execute floating-point (FP) instruction
  - How condition code bits are set on a move instruction
  - Size of the instruction cache
  - Type of floating-point format

ISA Analogy – Interface vs. Implementation

- **Intel64 ISA book**
  - http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html
- **ARMv8 ISA book (registration required)**
  - http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.set.architecture/index.html

Architecture

> **"the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation"**
>
> **-- *Amdahl, Blaauw, and Brooks, Architecture of the IBM System/360, IBM Journal of Research and Development, April 1964.***

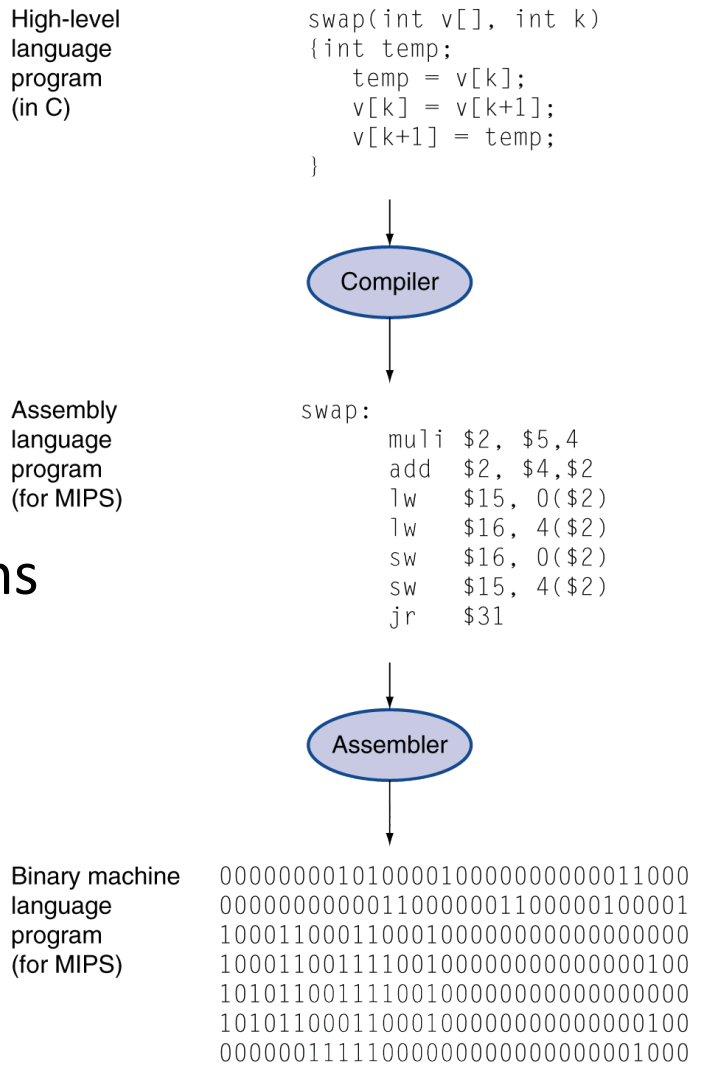
- **The visible interface between software and hardware**
- **What the user (OS, compiler, ...) needs to know to reason about how the machine behaves**
- **Abstracted from the details of how it may accomplish its task**

Architecture

- **Computer "Architecture" defines**
  - Instructions
    - Instruction set
    - Operand types
    - Data types (integers, FPs, ...)
    - Memory addressing modes, ...
  - Registers and other state
  - The interrupt/exception model
  - Memory management and protection
  - Virtual and physical address layout
  - I/O model
  - ...

Levels of Program Code

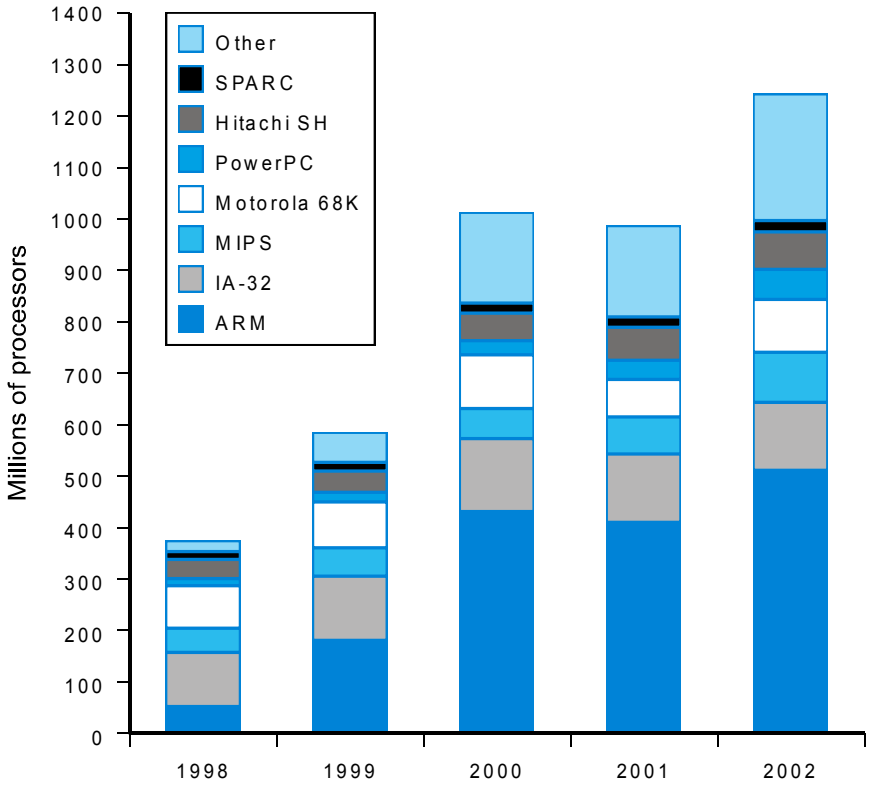
- **High-level language**
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- **Assembly language**
  - Textual representation of instructions
- **Hardware representation**
  - Binary digits (bits)
  - Encoded instructions and data



Operations of Computer Hardware

Instruction Set

- **The repertoire of instructions of a computer**
- **Different computers have different instruction sets**
  - But with many aspects in common
- **Early computers had very simple instruction sets**
  - Simplified implementation
- **Many modern computers also have simple instruction sets**
  - So-called “RISC” (Reduced Instruction Set Computer
  - e.g., MIPS, ARM, PowerPC, etc.



The MIPS Instruction Set

- **Used as the example throughout the course**
- **Stanford MIPS commercialized by MIPS Technologies (www.mips.com)**
- **Large share of embedded core market**
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …
  - Almost 100 million MIPS processors manufactured in 2002
  - Used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, ...
- **Typical of many modern ISAs**
  - See MIPS Reference Data tear-out card (“green card”)

The MIPS ISA: MIPS Green Card

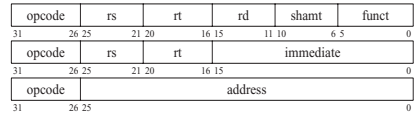
MIPS Reference Data ①

CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Add	add	R	R[rd] = R[rs] + R[rt]	(1)	0 / 20_{hex}
Add Immediate	addi	I	R[rt] = R[rs] + SignExtImm	(1,2)	8_{hex}
Add Imm. Unsigned	addiu	I	R[rt] = R[rs] + SignExtImm	(2)	9_{hex}
Add Unsigned	addu	R	R[rd] = R[rs] + R[rt]		0 / 21_{hex}
And	and	R	R[rd] = R[rs] & R[rt]		0 / 24_{hex}
And Immediate	andi	I	R[rt] = R[rs] & ZeroExtImm	(3)	c_{hex}
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr	(4)	5_{hex}
Jump	j	J	PC=JumpAddr	(5)	2_{hex}
Jump And Link	jal	J	R[31]=PC+8;PC=JumpAddr	(5)	3_{hex}
Jump Register	jr	R	PC=R[rs]		0 / 08_{hex}
Load Byte Unsigned	lbu	I	R[rt]={24'b0,M[R[rs]+SignExtImm](7:0)}	(2)	24_{hex}
Load Halfword Unsigned	lhu	I	R[rt]={16'b0,M[R[rs]+SignExtImm](15:0)}	(2)	25_{hex}
Load Linked	ll	I	R[rt] = M[R[rs]+SignExtImm]	(2,7)	30_{hex}
Load Upper Imm.	lui	I	R[rt] = {imm, 16'b0}		f_{hex}
Load Word	lw	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Nor	nor	R	R[rd] = ~ (R[rs] | R[rt])		0 / 27_{hex}
Or	or	R	R[rd] = R[rs] | R[rt]		0 / 25_{hex}
Or Immediate	ori	I	R[rt] = R[rs] | ZeroExtImm	(3)	d_{hex}
Set Less Than	slt	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0		0 / $2a_{hex}$
Set Less Than Imm.	slti	I	R[rt] = (R[rs] < SignExtImm)? 1 : 0	(2)	a_{hex}
Set Less Than Imm. Unsigned	sltiu	I	R[rt] = (R[rs] < SignExtImm) ? 1 : 0	(2,6)	b_{hex}
Set Less Than Unsig.	sltu	R	R[rd] = (R[rs] < R[rt]) ? 1 : 0	(6)	0 / $2b_{hex}$
Shift Left Logical	sll	R	R[rd] = R[rt] << shamt		0 / 00_{hex}
Shift Right Logical	srl	R	R[rd] = R[rt] >> shamt		0 / 02_{hex}
Store Byte	sb	I	M[R[rs]+SignExtImm](7:0) = R[rt](7:0)	(2)	28_{hex}
Store Conditional	sc	I	M[R[rs]+SignExtImm] = R[rt]; R[rt] = (*atomic*) ? 1 : 0	(2,7)	38_{hex}
Store Halfword	sh	I	M[R[rs]+SignExtImm](15:0) = R[rt](15:0)	(2)	29_{hex}
Store Word	sw	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	0 / 22_{hex}
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		0 / 23_{hex}

(1) May cause overflow exception
(2) SignExtImm = { 16{immediate[15]}, immediate }
(3) ZeroExtImm = { 16{1b'0}, immediate }
(4) BranchAddr = { 14{immediate[15]}, immediate, 2'b0 }
(5) JumpAddr = { PC+4[31:28], address, 2'b0 }
(6) Operands considered unsigned numbers (vs. 2's comp.)
(7) Atomic test&set pair; R[rt] = 1 if pair atomic, 0 if not atomic

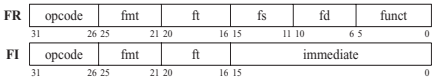
BASIC INSTRUCTION FORMATS



ARITHMETIC CORE INSTRUCTION SET ②

NAME, MNEMONIC		FOR-MAT	OPERATION		OPCODE / FMT /FT / FUNCT (Hex)
Branch On FP True	bc1t	FI	if(FPcond)PC=PC+4+BranchAddr	(4)	11/8/1/--
Branch On FP False	bc1f	FI	if(!FPcond)PC=PC+4+BranchAddr	(4)	11/8/0/--
Divide	div	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]		0/--/--/1a
Divide Unsigned	divu	R	Lo=R[rs]/R[rt]; Hi=R[rs]%R[rt]	(6)	0/--/--/1b
FP Add Single	add.s	FR	F[fd]= F[fs] + F[ft]		11/10/--/0
FP Add Double	add.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} + {F[ft],F[ft+1]}		11/11/--/0
FP Compare Single	c.x.s*	FR	FPcond = (F[fs] *op* F[ft]) ? 1 : 0		11/10/--/*y*
FP Compare Double	c.x.d*	FR	FPcond = ({F[fs],F[fs+1]} *op* {F[ft],F[ft+1]}) ? 1 : 0		11/11/--/*y*
FP Divide Single	div.s	FR	F[fd] = F[fs] / F[ft]		11/10/--/3
FP Divide Double	div.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} / {F[ft],F[ft+1]}		11/11/--/3
FP Multiply Single	mul.s	FR	F[fd] = F[fs] * F[ft]		11/10/--/2
FP Multiply Double	mul.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} * {F[ft],F[ft+1]}		11/11/--/2
FP Subtract Single	sub.s	FR	F[fd]=F[fs] - F[ft]		11/10/--/1
FP Subtract Double	sub.d	FR	{F[fd],F[fd+1]} = {F[fs],F[fs+1]} - {F[ft],F[ft+1]}		11/11/--/1
Load FP Single	lwc1	I	F[rt]=M[R[rs]+SignExtImm]	(2)	31/--/--/--
Load FP Double	ldc1	I	F[rt]=M[R[rs]+SignExtImm]; F[rt+1]=M[R[rs]+SignExtImm+4]	(2)	35/--/--/--
Move From Hi	mfhi	R	R[rd] = Hi		0 /--/--/10
Move From Lo	mflo	R	R[rd] = Lo		0 /--/--/12
Move From Control	mfc0	R	R[rd] = CR[rs]		10 /0/--/0
Multiply	mult	R	{Hi,Lo} = R[rs] * R[rt]		0/--/--/18
Multiply Unsigned	multu	R	{Hi,Lo} = R[rs] * R[rt]	(6)	0/--/--/19
Shift Right Arith.	sra	R	R[rd] = R[rt] >>> shamt		0/--/--/3
Store FP Single	swc1	I	M[R[rs]+SignExtImm] = F[rt]	(2)	39/--/--/--
Store FP Double	sdc1	I	M[R[rs]+SignExtImm] = F[rt]; M[R[rs]+SignExtImm+4] = F[rt+1]	(2)	3d/--/--/--

FLOATING-POINT INSTRUCTION FORMATS



PSEUDOINSTRUCTION SET

NAME	MNEMONIC	OPERATION
Branch Less Than	blt	if(R[rs]<R[rt]) PC = Label
Branch Greater Than	bgt	if(R[rs]>R[rt]) PC = Label
Branch Less Than or Equal	ble	if(R[rs]<=R[rt]) PC = Label
Branch Greater Than or Equal	bge	if(R[rs]>=R[rt]) PC = Label
Load Immediate	li	R[rd] = immediate
Move	move	R[rd] = R[rs]

REGISTER NAME, NUMBER, USE, CALL CONVENTION

NAME	NUMBER	USE	PRESERVED ACROSS A CALL?
$zero	0	The Constant Value 0	N.A.
$at	1	Assembler Temporary	No
$v0-$v1	2-3	Values for Function Results and Expression Evaluation	No
$a0-$a3	4-7	Arguments	No
$t0-$t7	8-15	Temporaries	No
$s0-$s7	16-23	Saved Temporaries	Yes
$t8-$t9	24-25	Temporaries	No
$k0-$k1	26-27	Reserved for OS Kernel	No
$gp	28	Global Pointer	Yes
$sp	29	Stack Pointer	Yes
$fp	30	Frame Pointer	Yes
$ra	31	Return Address	Yes

OPCODES, BASE CONVERSION, ASCII SYMBOLS ③

MIPS opcode (31:26)	(1) MIPS funct (5:0)	(2) MIPS funct (5:0)	Binary	Deci-mal	Hexa-deci-mal	ASCII Char-acter	Deci-mal	Hexa-deci-mal	ASCII Char-acter
(1)	sll	add.*f*	00 0000	0	0	NUL	64	40	@
		sub.*f*	00 0001	1	1	SOH	65	41	A
j	srl	mul.*f*	00 0010	2	2	STX	66	42	B
jal	sra	div.*f*	00 0011	3	3	ETX	67	43	C
beq	sllv	sqrt.*f*	00 0100	4	4	EOT	68	44	D
bne		abs.*f*	00 0101	5	5	ENQ	69	45	E
blez	srlv	mov.*f*	00 0110	6	6	ACK	70	46	F
bgtz	srav	neg.*f*	00 0111	7	7	BEL	71	47	G
addi	jr		00 1000	8	8	BS	72	48	H
addiu	jalr		00 1001	9	9	HT	73	49	I
slti	movz		00 1010	10	a	LF	74	4a	J
sltiu	movn		00 1011	11	b	VT	75	4b	K
andi	syscall	round.*w.f*	00 1100	12	c	FF	76	4c	L
ori	break	trunc.*w.f*	00 1101	13	d	CR	77	4d	M
xori		ceil.*w.f*	00 1110	14	e	SO	78	4e	N
lui	sync	floor.*w.f*	00 1111	15	f	SI	79	4f	O
	mfhi		01 0000	16	10	DLE	80	50	P
(2)	mthi		01 0001	17	11	DC1	81	51	Q
	mflo	movz.*f*	01 0010	18	12	DC2	82	52	R
	mtlo	movn.*f*	01 0011	19	13	DC3	83	53	S
			01 0100	20	14	DC4	84	54	T
			01 0101	21	15	NAK	85	55	U
			01 0110	22	16	SYN	86	56	V
			01 0111	23	17	ETB	87	57	W
	mult		01 1000	24	18	CAN	88	58	X
	multu		01 1001	25	19	EM	89	59	Y
	div		01 1010	26	1a	SUB	90	5a	Z
	divu		01 1011	27	1b	ESC	91	5b	[
			01 1100	28	1c	FS	92	5c	\
			01 1101	29	1d	GS	93	5d	]
			01 1110	30	1e	RS	94	5e	^
			01 1111	31	1f	US	95	5f	_
lb	add	cvt.s.*f*	10 0000	32	20	Space	96	60	`
lh	addu	cvt.d.*f*	10 0001	33	21	!	97	61	a
lwl	sub		10 0010	34	22	"	98	62	b
lw	subu		10 0011	35	23	#	99	63	c
lbu	and	cvt.w.*f*	10 0100	36	24	$	100	64	d
lhu	or		10 0101	37	25	%	101	65	e
lwr	xor		10 0110	38	26	&	102	66	f
	nor		10 0111	39	27	'	103	67	g
sb			10 1000	40	28	(	104	68	h
sh			10 1001	41	29	)	105	69	i
swl	slt		10 1010	42	2a	*	106	6a	j
sw	sltu		10 1011	43	2b	+	107	6b	k
			10 1100	44	2c	,	108	6c	l
			10 1101	45	2d	-	109	6d	m
swr			10 1110	46	2e	.	110	6e	n
cache			10 1111	47	2f	/	111	6f	o
ll	tge	c.*f.f*	11 0000	48	30	0	112	70	p
lwc1	tgeu	c.un.*f*	11 0001	49	31	1	113	71	q
lwc2	tlt	c.eq.*f*	11 0010	50	32	2	114	72	r
pref	tltu	c.ueq.*f*	11 0011	51	33	3	115	73	s
	teq	c.olt.*f*	11 0100	52	34	4	116	74	t
ldc1		c.ult.*f*	11 0101	53	35	5	117	75	u
ldc2	tne	c.ole.*f*	11 0110	54	36	6	118	76	v
		c.ule.*f*	11 0111	55	37	7	119	77	w
sc		c.sf.*f*	11 1000	56	38	8	120	78	x
swc1		c.ngle.*f*	11 1001	57	39	9	121	79	y
swc2		c.seq.*f*	11 1010	58	3a	:	122	7a	z
		c.ngl.*f*	11 1011	59	3b	;	123	7b	{
		c.lt.*f*	11 1100	60	3c	<	124	7c	|
sdc1		c.nge.*f*	11 1101	61	3d	=	125	7d	}
sdc2		c.le.*f*	11 1110	62	3e	>	126	7e	~
		c.ngt.*f*	11 1111	63	3f	?	127	7f	DEL

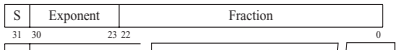
(1) opcode(31:26) == 0
(2) opcode(31:26) == 17_{ten} (11_{hex}); if fmt(25:21)==16_{ten} (10_{hex}) *f* = s (single); if fmt(25:21)==17_{ten} (11_{hex}) *f* = d (double)

IEEE 754 FLOATING-POINT STANDARD

$(-1)^S \times (1 + \text{Fraction}) \times 2^{(\text{Exponent} - \text{Bias})}$

where Single Precision Bias = 127, Double Precision Bias = 1023.

IEEE Single Precision and Double Precision Formats:



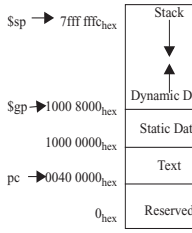
④

IEEE 754 Symbols

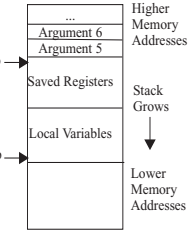
Exponent	Fraction	Object
0	0	± 0
0	≠0	± Denorm
1 to MAX - 1	anything	± Fl. Pt. Num.
MAX	0	±∞
MAX	≠0	NaN

S.P. MAX = 255, D.P. MAX = 2047

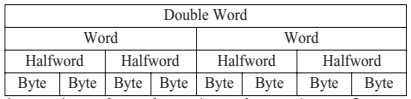
MEMORY ALLOCATION



STACK FRAME

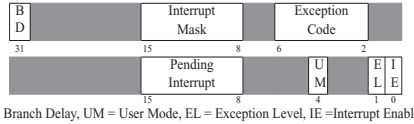


DATA ALIGNMENT



Value of three least significant bits of byte address (Big Endian)

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS



BD = Branch Delay, UM = User Mode, EL = Exception Level, IE =Interrupt Enable

EXCEPTION CODES

Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on Instruction Fetch	12	Ov	Arithmetic Overflow Exception
7	DBE	Bus Error on Load or Store	13	Tr	Trap
8	Sys	Syscall Exception	15	FPE	Floating Point Exception

SIZE PREFIXES (10^X for Disk, Communication; 2^X for Memory)

SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX	SIZE	PRE-FIX
$10^3, 2^{10}$	Kilo-	$10^{15}, 2^{50}$	Peta-	10^{-3}	milli-	10^{-15}	femto-
$10^6, 2^{20}$	Mega-	$10^{18}, 2^{60}$	Exa-	10^{-6}	micro-	10^{-18}	atto-
$10^9, 2^{30}$	Giga-	$10^{21}, 2^{70}$	Zetta-	10^{-9}	nano-	10^{-21}	zepto-
$10^{12}, 2^{40}$	Tera-	$10^{24}, 2^{80}$	Yotta-	10^{-12}	pico-	10^{-24}	yocto-

The symbol for each prefix is just its first letter, except μ is used for micro.

Arithmetic Operations

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

```
add a, b, c  # a gets b + c
```

- **All arithmetic operations have this form**

- ***Design Principle 1:*** **Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

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

- Compiled MIPS code (assembly):

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

Operands of Computer Hardware

Register Operands

- **Arithmetic instructions use register operands**
- **MIPS has a 32 × 32-bit register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- **Assembler names**
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables

- ***Design Principle 2:*** **Smaller is faster**
  - c.f. main memory: millions of locations

Register Operand Example

- **C code:**

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

  - Assume `f`, `g`, `h`, `i`, `j` are stored in $s0, $s1, $s2, $s3, $s4

- **Compiled MIPS code:**

Register Operands: MIPS Registers

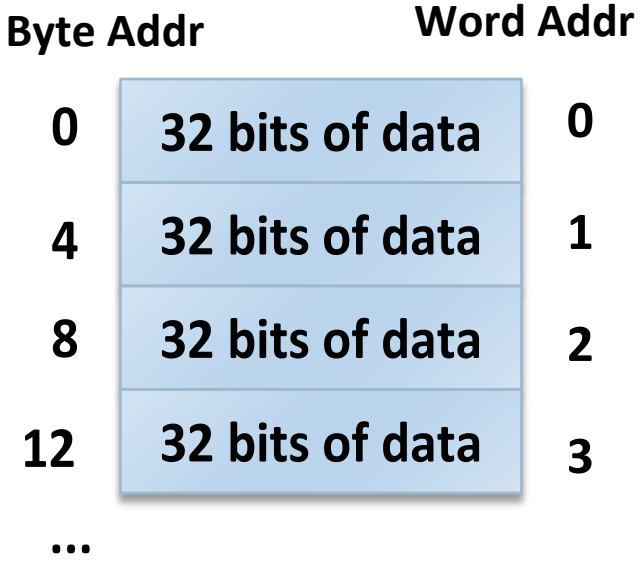
- **32 general-purpose registers**

#	Name	Usage
0	$zero	The constant value 0
1	$at	Assembler temporary
2	$v0	Values for results and expression evaluation
3	$v1	
4	$a0	Arguments
5	$a1	
6	$a2	
7	$a3	
8	$t0	Temporaries (Caller-save registers)
9	$t1	
10	$t2	
11	$t3	
12	$t4	
13	$t5	
14	$t6	
15	$t7	

#	Name	Usage
16	$s0	Saved temporaries (Callee-save registers)
17	$s1	
18	$s2	
19	$s3	
20	$s4	
21	$s5	
22	$s6	
23	$s7	
24	$t8	More temporaries (Caller-save registers)
25	$t9	
26	$k0	Reserved for OS kernel
27	$k1	
28	$gp	Global pointer
29	$sp	Stack pointer
30	$fp	Frame pointer
31	$ra	Return address

Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into register
  - Store result from register to memory
- **Memory is byte addressed**
  - Each address identifies an 8-bit byte
- **Words are aligned in memory**
  - Address must be a multiple of 4
- **MIPS is Big Endian**
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address



Memory Operand Example 1

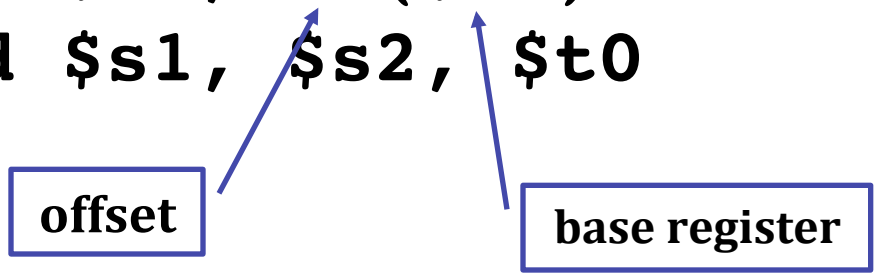
- **C code:**

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

  - g in $s1, h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  - 4 bytes per word
  - Index 8 requires offset of 32

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



Memory Operand Example 2

- **C code:**

```
A[12] = h + A[4];
```

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

- **Compiled MIPS code:**
  - Index 4 requires offset of 16
    (Again, 4 bytes per word)

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

Registers vs. Memory

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

Immediate Operands

- **Constant data specified in an instruction**

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

- **No subtract immediate instruction**
  - Just use a negative constant

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

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

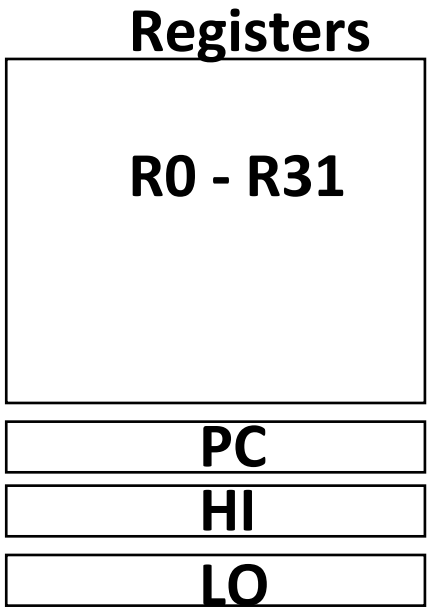
The Constant Zero

- **MIPS register 0 ($zero) is the constant 0**
  - Cannot be overwritten (i.e., read-only)
- **Useful for common operations**
  - E.g., move between registers

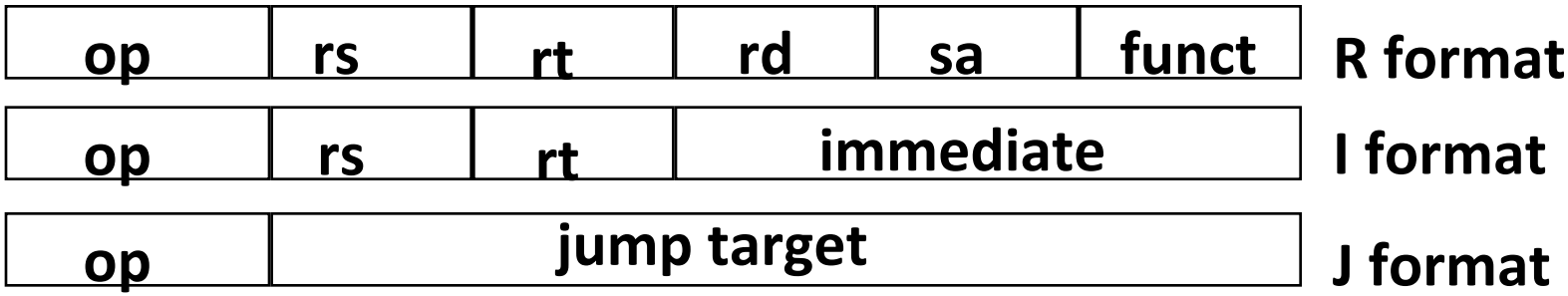
```
add $t2, $s1, $zero
```

MIPS-32 ISA

- **Instruction Categories**
  - Computational
  - Load/Store
  - Jump and Branch
  - Floating Point
    - coprocessor
  - Memory Management
  - Special



- **3 Instruction Formats: all 32 bits wide**



Logical Operations

Logical Operations

- **Instructions for bitwise manipulation**

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

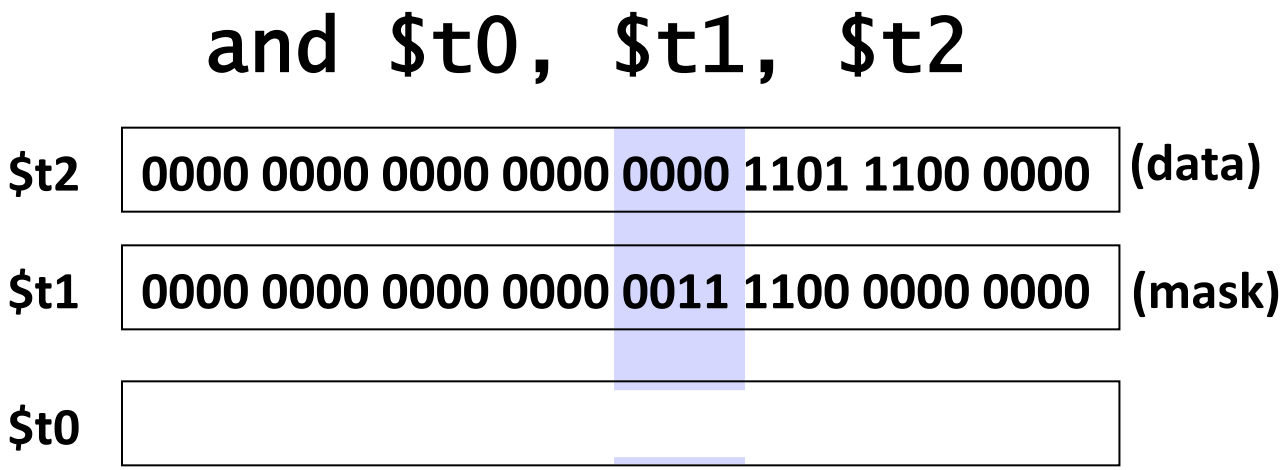
Useful for extracting and inserting groups of bits in a word

Shift Operations

- **shamt: how many positions to shift**
- **Shift left logical**
  - Shift left and fill with 0 bits
  - `sll` by *i* bits multiplies by 2^i
- **Shift right logical**
  - Shift right and fill with 0 bits
  - `srl` by *i* bits divides by 2^i (unsigned only)
  - cf. `sra`: shift right arithmetic

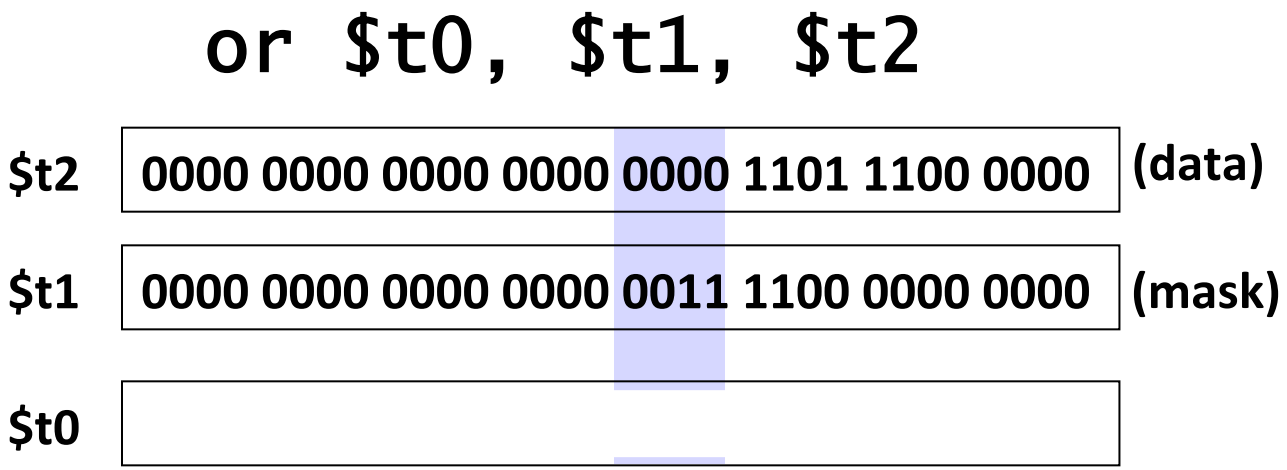
AND Operations

- **Useful to mask bits in a word**
  - Select some bits, clear others to 0



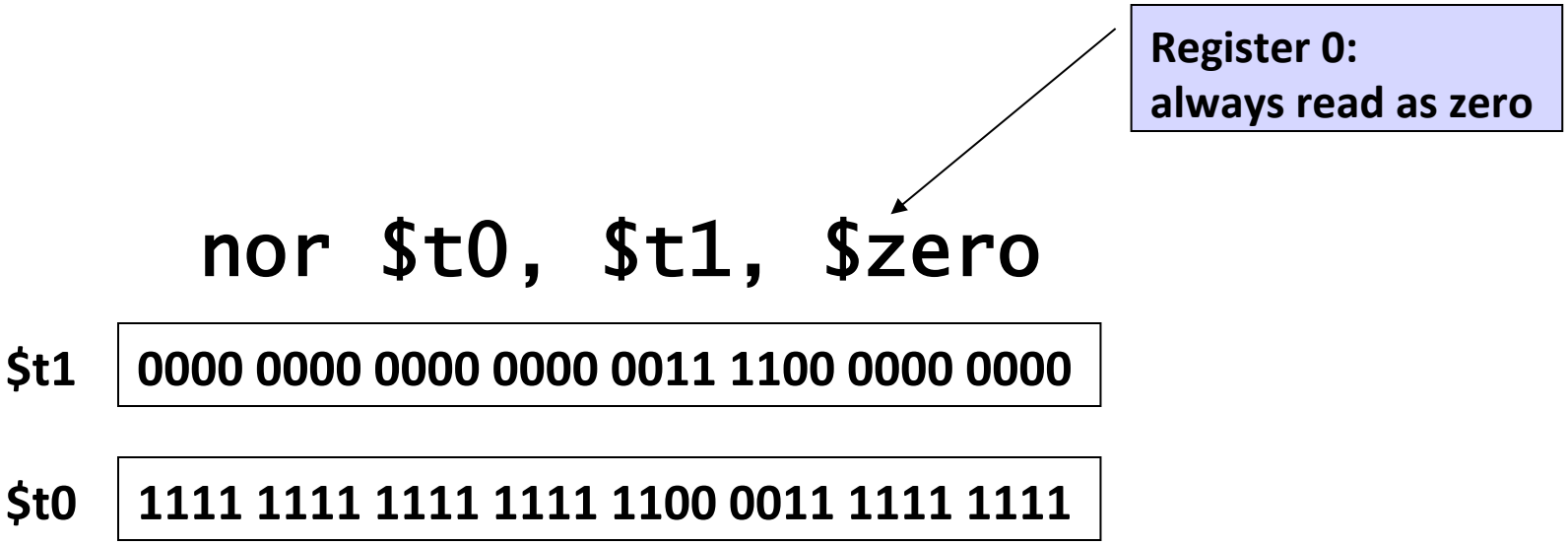
OR Operations

- **Useful to include bits in a word**
  - Set some bits to 1, leave others unchanged



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)



Instructions for Making Decisions

Conditional Operations

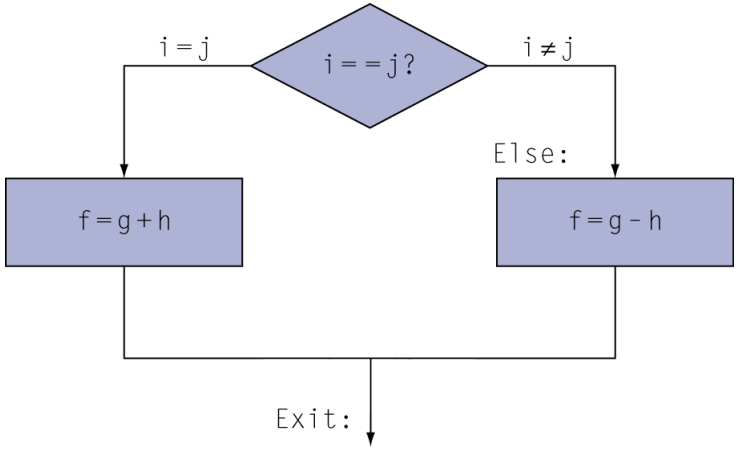
- **Branch to a labeled instruction if a condition is true**
  - Otherwise, continue sequentially
- `beq rs, rt, L1`
  - if (rs == rt) branch to instruction labeled L1;
- `bne rs, rt, L1`
  - if (rs != rt) branch to instruction labeled L1;
- `j L1`
  - unconditional jump to instruction labeled L1

Compiling If Statements

- **C code:**

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

  - f, g, … in $s0, $s1, …
- **Compiled MIPS code:**



Assembler calculates addresses

Compiling Loop Statements

- **C code:**

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

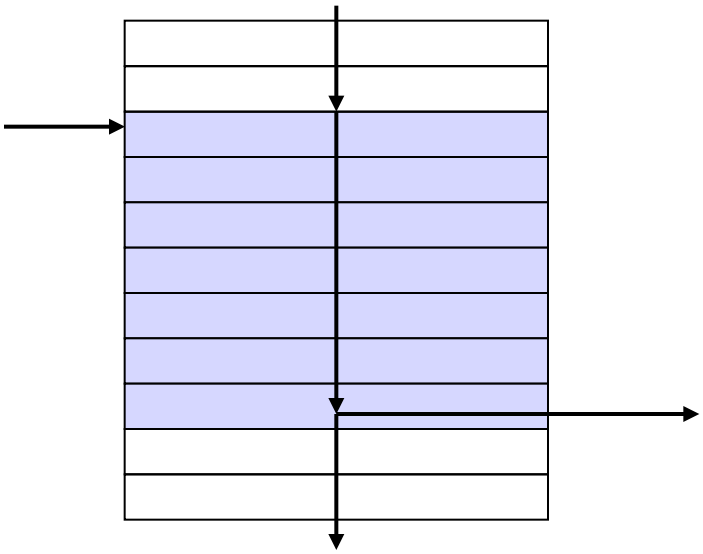
  - i in $s3, k in $s5, address of save in $s6

- **Compiled MIPS code:**

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

Basic Blocks

- **A basic block is a sequence of instructions with**
  - No embedded branches (except at end)
  - No branch targets (except at beginning)
- **A compiler identifies basic blocks for optimization**
- **An advanced processor can accelerate execution of basic blocks**



More Conditional Operations

- **Set result to 1 if a condition is true**
  - Otherwise, set to 0
- `slt rd, rs, rt`
  - 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
```

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**

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

Communicating with People (Handling Characters)

Character Data

- **Byte-encoded character sets**
  - ASCII: 128 characters
    - 95 graphic, 33 control
  - Latin-1: 256 characters
    - ASCII, +96 more graphic characters
- **Unicode: 32-bit character set**
  - Used in Java, C++ wide characters, …
  - Most of the world's alphabets, plus symbols
  - UTF-8, UTF-16: variable-length encodings

Byte/Halfword Operations

- **Could use bitwise operations**
- **MIPS byte/halfword load/store**
  - String processing is a common case

```
lb rt, offset(rs)      lh rt, offset(rs)
```

  - Sign extend to 32 bits in rt

```
lbu rt, offset(rs)     lhu rt, offset(rs)
```

  - Zero extend to 32 bits in rt

```
sb rt, offset(rs)      sh rt, offset(rs)
```

  - Store just rightmost byte/halfword

String Copy Example

- **C code (naïve):**
  - Null-terminated string

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

  - Addresses of x, y in $a0, $a1
  - i in $s0

String Copy Example

- **MIPS code:**

```
strcpy:
    addi $sp, $sp, -4       # adjust stack for 1 item
    sw   $s0, 0($sp)        # save $s0
    add  $s0, $zero, $zero  # i = 0
L1: add  $t1, $s0, $a1      # addr of y[i] in $t1
    lbu  $t2, 0($t1)        # $t2 = y[i]
    add  $t3, $s0, $a0      # addr of x[i] in $t3
    sb   $t2, 0($t3)        # x[i] = y[i]
    beq  $t2, $zero, L2     # exit loop if y[i] == 0
    addi $s0, $s0, 1        # i = i + 1
    j    L1                 # next iteration of loop
L2: lw   $s0, 0($sp)        # restore saved $s0
    addi $sp, $sp, 4        # pop 1 item from stack
    jr   $ra                # and return
```

Summary

- **Now you understand the language of CPU: instructions**
  - Arithmetic and logical
  - Memory access (loads and stores)
  - Making decisions (branches)
  - Byte/halfword operations

- **Next time we will learn how to implement CPU that can execute these instructions!**