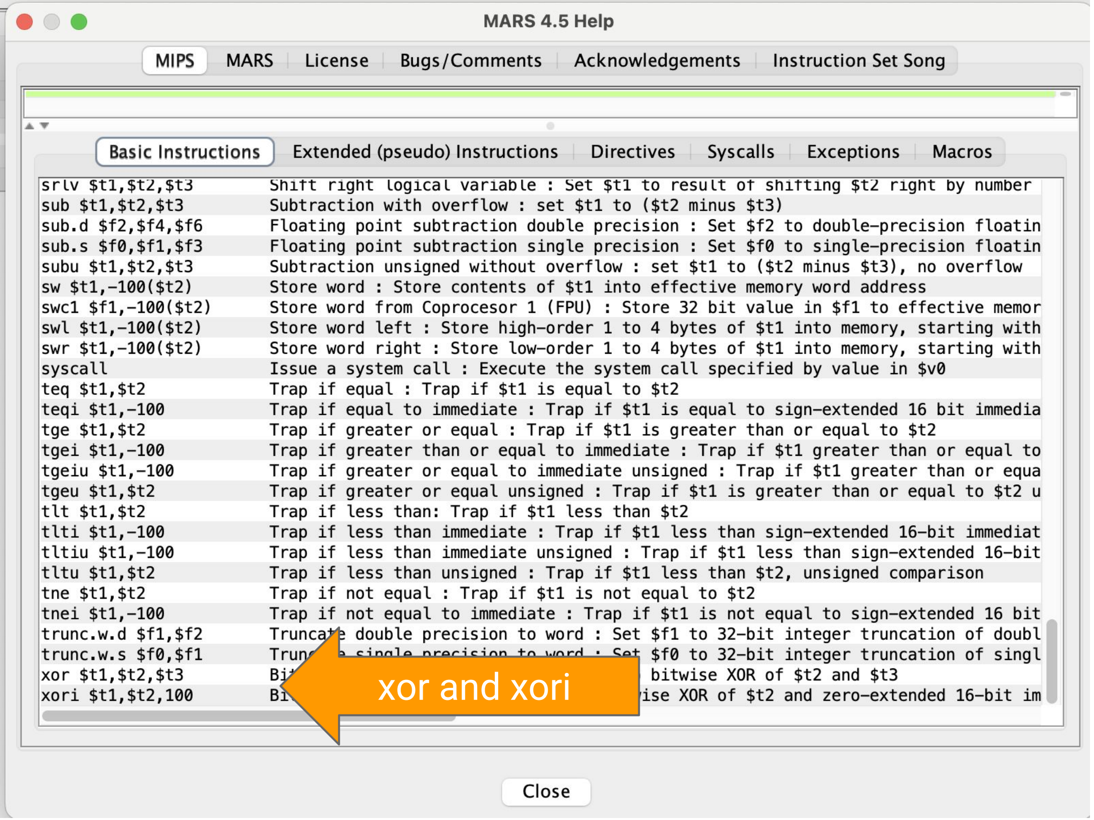
CS 2340 - Computer Architecture

8 System Software, Machine Language - Part 2
Dr. Alice Wang

Q. How does a computer eat chips?
A. With Mega-bytes

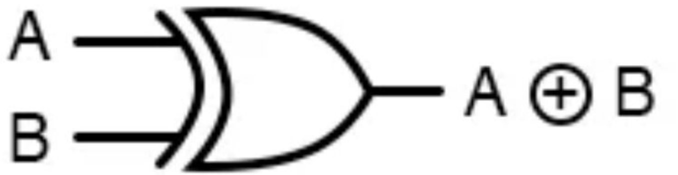
Research

- Does MIPS have an XOR function? Yes!



```
srlv $t1,$t2,$t3        Shift right logical variable : Set $t1 to result of shifting $t2 right by number
sub $t1,$t2,$t3         Subtraction with overflow : set $t1 to ($t2 minus $t3)
sub.d $f2,$f4,$f6       Floating point subtraction double precision : Set $f2 to double-precision floatin
sub.s $f0,$f1,$f3       Floating point subtraction single precision : Set $f0 to single-precision floatin
subu $t1,$t2,$t3        Subtraction unsigned without overflow : set $t1 to ($t2 minus $t3), no overflow
sw $t1,-100($t2)        Store word : Store contents of $t1 into effective memory word address
swc1 $f1,-100($t2)      Store word from Coprocesor 1 (FPU) : Store 32 bit value in $f1 to effective memor
swl $t1,-100($t2)       Store word left : Store high-order 1 to 4 bytes of $t1 into memory, starting with
swr $t1,-100($t2)       Store word right : Store low-order 1 to 4 bytes of $t1 into memory, starting with
syscall                 Issue a system call : Execute the system call specified by value in $v0
teq $t1,$t2             Trap if equal : Trap if $t1 is equal to $t2
teqi $t1,-100           Trap if equal to immediate : Trap if $t1 is equal to sign-extended 16 bit immedia
tge $t1,$t2             Trap if greater or equal : Trap if $t1 is greater than or equal to $t2
tgei $t1,-100           Trap if greater than or equal to immediate : Trap if $t1 greater than or equal to
tgeiu $t1,-100          Trap if greater or equal to immediate unsigned : Trap if $t1 greater than or equa
tgeu $t1,$t2            Trap if greater or equal unsigned : Trap if $t1 is greater than or equal to $t2 u
tlt $t1,$t2             Trap if less than: Trap if $t1 less than $t2
tlti $t1,-100           Trap if less than immediate : Trap if $t1 less than sign-extended 16-bit immediat
tltiu $t1,-100          Trap if less than immediate unsigned : Trap if $t1 less than sign-extended 16-bit
tltu $t1,$t2            Trap if less than unsigned : Trap if $t1 less than $t2, unsigned comparison
tne $t1,$t2             Trap if not equal : Trap if $t1 is not equal to $t2
tnei $t1,-100           Trap if not equal to immediate : Trap if $t1 is not equal to sign-extended 16 bit
trunc.w.d $f1,$f2       Truncate double precision to word : Set $f1 to 32-bit integer truncation of doubl
trunc.w.s $f0,$f1       Trun    e single precision to word : Set $f0 to 32-bit integer truncation of singl
xor $t1,$t2,$t3         Bi                 bitwise XOR of $t2 and $t3
xori $t1,$t2,100        Bi                ise XOR of $t2 and zero-extended 16-bit im
```

xor and xori



XOR is true if and only if the inputs differ (one is true, one is false).

A	B	Y
0	0	0
0	1	1
1	0	1
1	1	0

Research

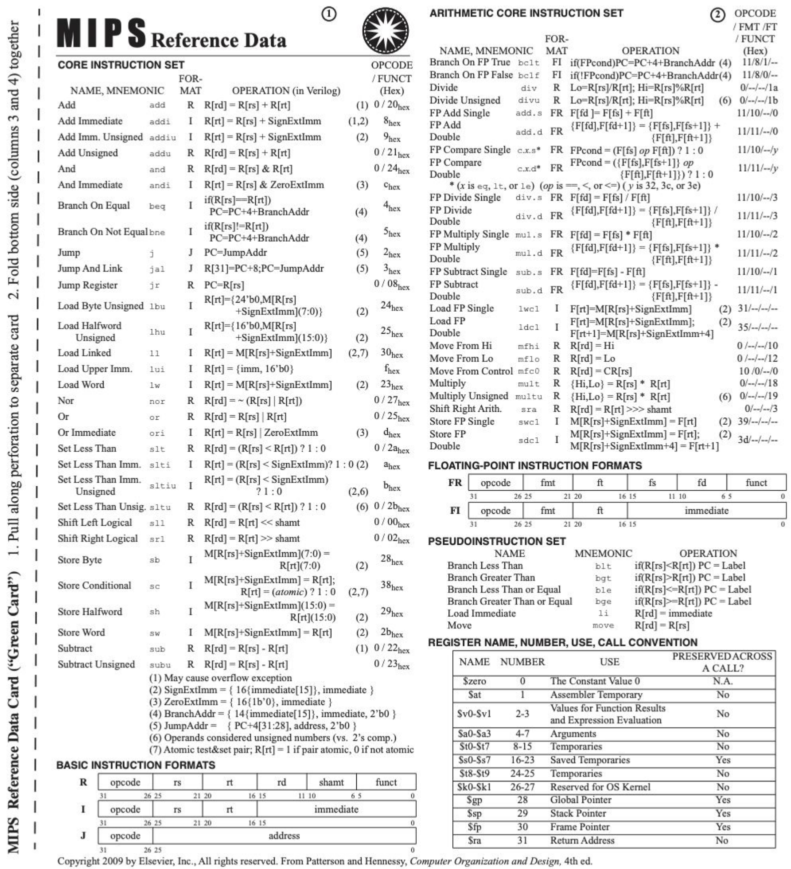
- What are some real-life ways to use OR in MIPS?
- Setting bits to “1” or bit-masking
- Example:
    - $t0 = 0b0110_0001
    - You want to set bit 3 of $t0 to “1”, leaving the rest of the bits unchanged
    - **`or $t0, $t0, 0x0000_1000`**
- $t0 could be enable bits, flags

Review of Last Lecture

- Logical Operations
  - `and, or, not`
- Shifters
  - `sll, srl, sra`
- MIPS has 3 instruction formats
  - R-, I- and J- type
  - Last time R-type, This time I- and J- type
- More Examples for practice
  - MIPS interactive assembler, More exercises

MIPS Reference Card – pdf on elearning

you can bring this to the exam



MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

① **MIPS** Reference Data

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

R	opcode	rs	rt	rd	shamt	funct
	31 26	25 21	20 16	15 11	10 6	5 0
I	opcode	rs	rt	immediate		
	31 26	25 21	20 16	15 0		
J	opcode	address				
	31 26	25 0				

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Computer Organization and Design*, 4th ed.

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*
* (*x* is eq, lt, or le) (*op* is ==, <, or <=) (*y* is 32, 3c, or 3e)					
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

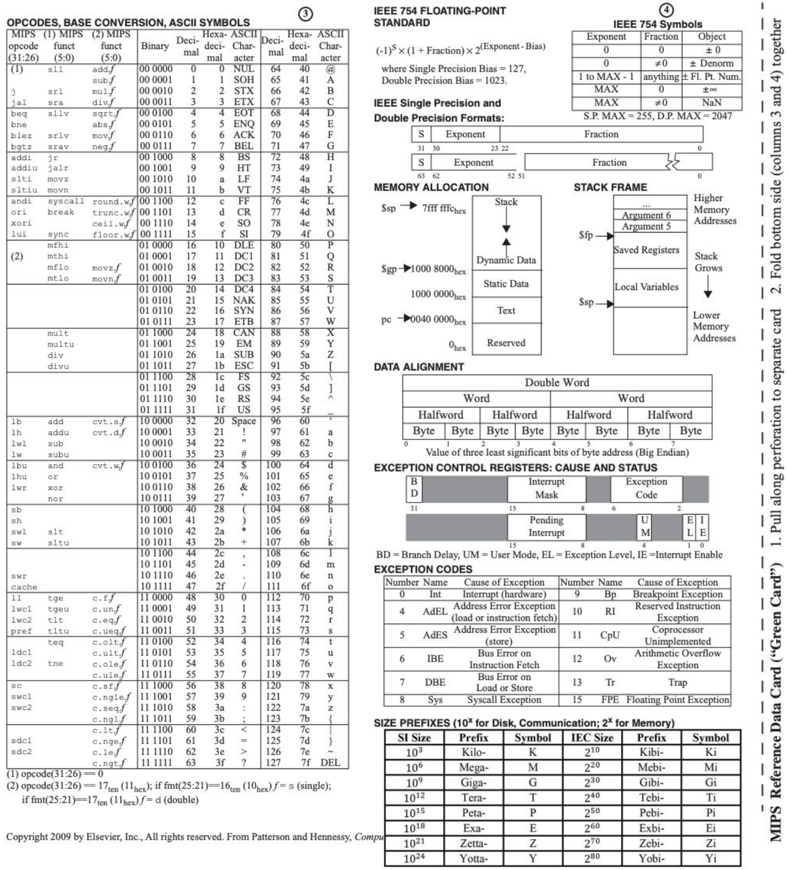
FR	opcode	fmt	ft	fs	fd	funct
	31 26	25 21	20 16	15 11	10 6	5 0
FI	opcode	fmt	ft	immediate		
	31 26	25 21	20 16	15 0		

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	No



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)

Copyright 2009 by Elsevier, Inc., All rights reserved. From Patterson and Hennessy, *Compu*

IEEE 754 FLOATING-POINT STANDARD ④

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

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

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

IEEE Single Precision and Double Precision Formats:

S	Exponent	Fraction
31	30 23	22 0
S	Exponent	Fraction
63	62 52	51 0

MEMORY ALLOCATION

$sp → 7fff fffc$_{hex}$ Stack
Dynamic Data
$gp → 1000 8000$_{hex}$ Static Data
1000 0000$_{hex}$ Text
pc → 0040 0000$_{hex}$ Reserved
0$_{hex}$

STACK FRAME

...
Argument 6
Argument 5
$fp → Saved Registers
Local Variables
$sp →

Higher Memory Addresses
Stack Grows
Lower Memory Addresses

DATA ALIGNMENT

Double Word							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0	1	2	3	4	5	6	7

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

EXCEPTION CONTROL REGISTERS: CAUSE AND STATUS

BD		Interrupt Mask		Exception Code	
31		15 8		6 2	
		Pending Interrupt		UM	EL IE
		15 8		4	1 0

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

EXCEPTION CODES

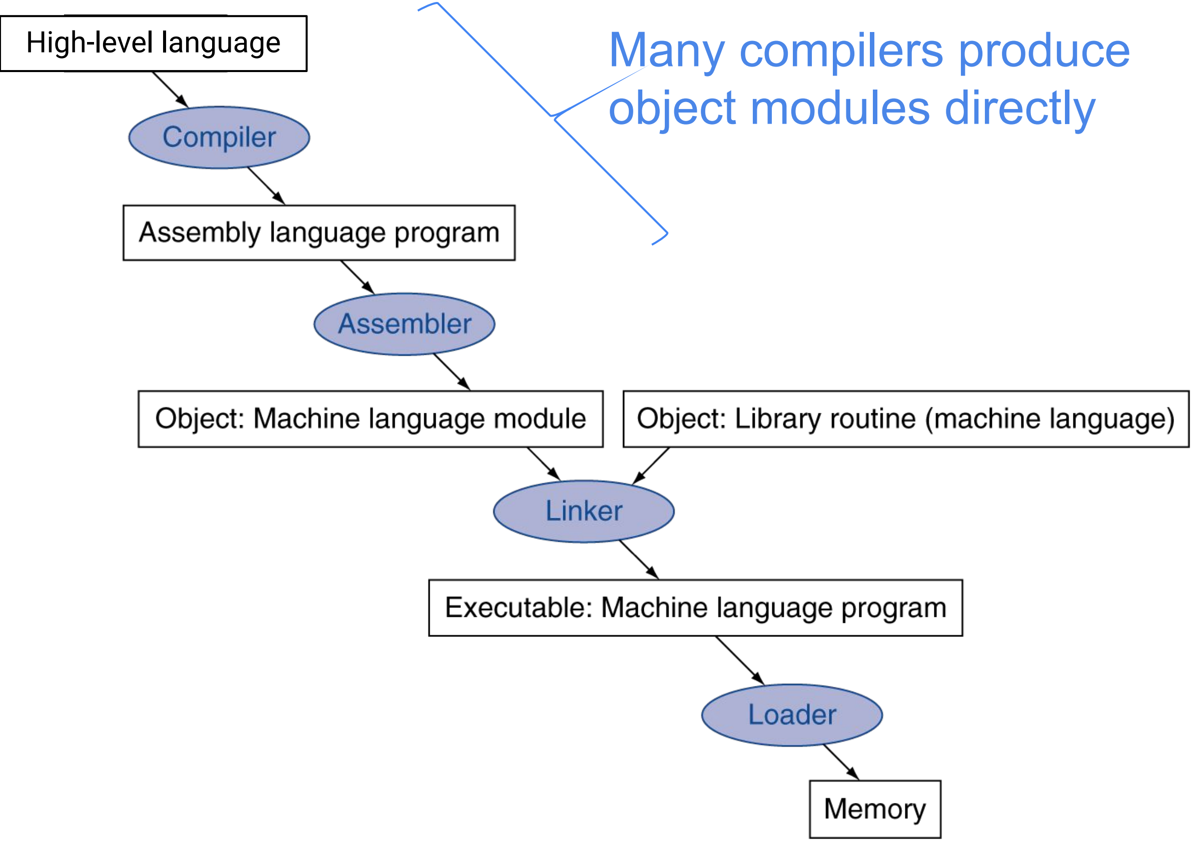
Number	Name	Cause of Exception	Number	Name	Cause of Exception
0	Int	Interrupt (hardware)	9	Bp	Breakpoint Exception
4	AdEL	Address Error Exception (load or instruction fetch)	10	RI	Reserved Instruction Exception
5	AdES	Address Error Exception (store)	11	CpU	Coprocessor Unimplemented
6	IBE	Bus Error on 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)

SI Size	Prefix	Symbol	IEC Size	Prefix	Symbol
10^3	Kilo-	K	2^{10}	Kibi-	Ki
10^6	Mega-	M	2^{20}	Mebi-	Mi
10^9	Giga-	G	2^{30}	Gibi-	Gi
10^{12}	Tera-	T	2^{40}	Tebi-	Ti
10^{15}	Peta-	P	2^{50}	Pebi-	Pi
10^{18}	Exa-	E	2^{60}	Exbi-	Ei
10^{21}	Zetta-	Z	2^{70}	Zebi-	Zi
10^{24}	Yotta-	Y	2^{80}	Yobi-	Yi

MIPS Reference Data Card ("Green Card") 1. Pull along perforation to separate card 2. Fold bottom side (columns 3 and 4) together

Review: Translation Hierarchy


High-level language
Compiler
Assembly language program
Assembler
Object: Machine language module
Object: Library routine (machine language)
Linker
Executable: Machine language program
Loader
Memory

Many compilers produce object modules directly

Static linking (there is also dynamic linking)

Assembler: Produces an Object Module

- Assembler translates assembly code program into machine instructions (0’s and 1’s)
- Provides information for building a complete program from the pieces. For an example UNIX system, 6 distinct pieces.
    - **Header:** described contents of object module
    - **Text segment:** translated instructions
    - **Static data segment:** data allocated for the life of the program
    - **Relocation info:** for contents that depend on absolute location of loaded program
    - **Symbol table:** global definitions and external references
    - **Debug info:** for associating with source code for a debugger

Static Linking

- Linker produces an executable image
  - Merges segments
  - Determines the address of labels
  - Patches location-dependent and external references
- Uses the relocation information and symbol table in each object to resolve undefined labels
  - For example in branches and jumps
- Static linking is done before execution
- Downsides of static linking
  - Library routines, which can be large, become part of the executable
  - Library routines won’t get updates or new versions

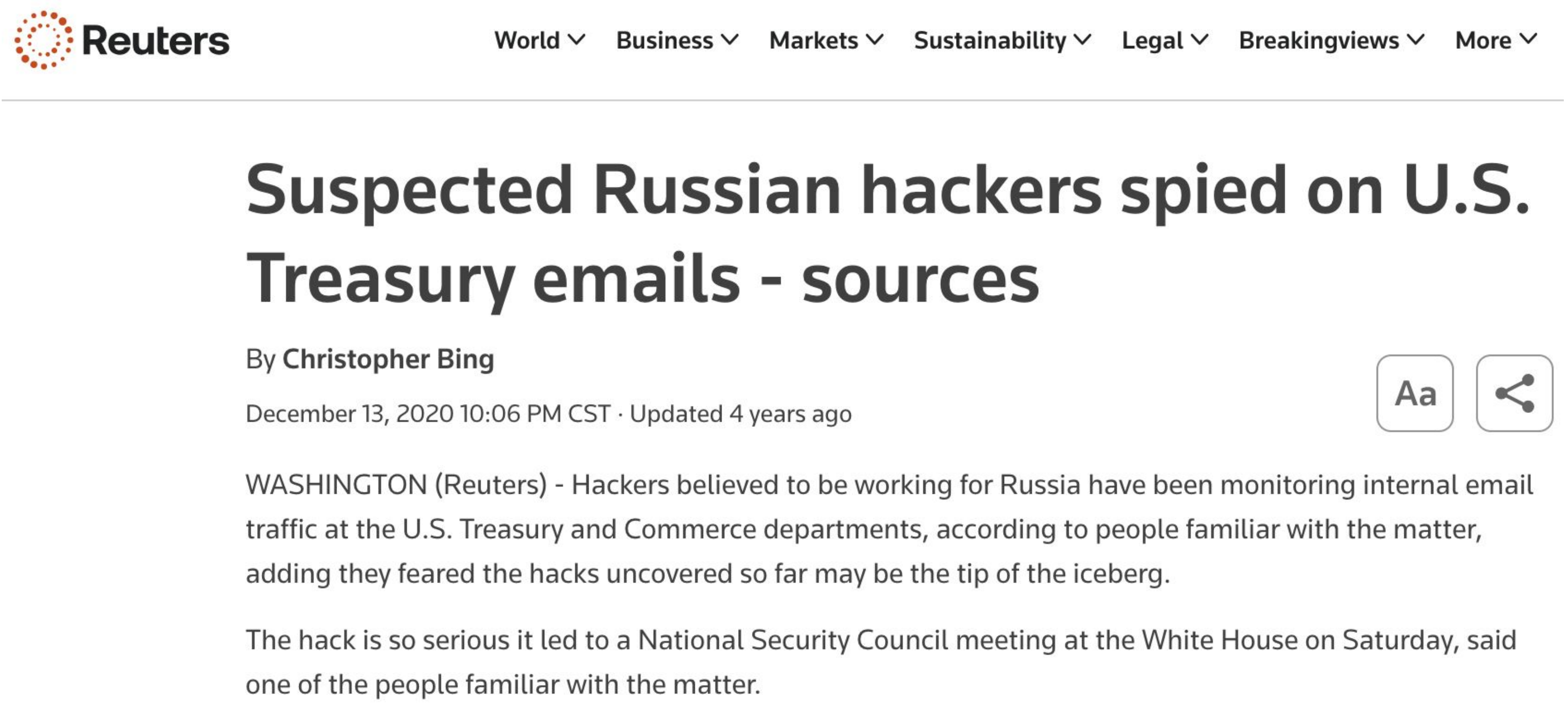
Dynamic Linking Libraries (DLL's)

- Dynamic Linking: only link/load library procedure when it is called during execution

- **Pro:** Avoids image bloat caused by static linking of all referenced libraries
- **Pro:** Automatically picks up new library versions

- **Cons:** Security can be compromised by inserting a harmful DLL. DLL's are particularly vulnerable to malware.
- **Cons:** Compatibility is a challenge. If you update or replace the DLL, and the code was based on the previous version, your program can break.

Solar Winds hack – a faulty DLL



Suspected Russian hackers spied on U.S. Treasury emails - sources

By **Christopher Bing**

December 13, 2020 10:06 PM CST · Updated 4 years ago

WASHINGTON (Reuters) - Hackers believed to be working for Russia have been monitoring internal email traffic at the U.S. Treasury and Commerce departments, according to people familiar with the matter, adding they feared the hacks uncovered so far may be the tip of the iceberg.

The hack is so serious it led to a National Security Council meeting at the White House on Saturday, said one of the people familiar with the matter.

Review: R-type – Instruction Example

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

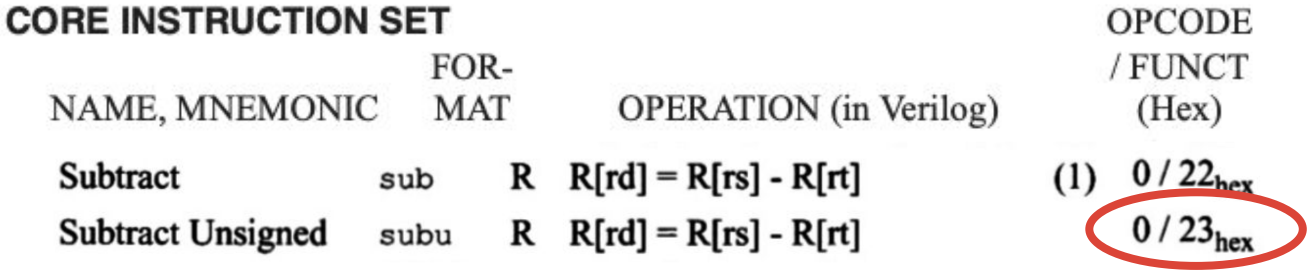
- All instructions that have register operands only
    - **`add, sub, and, or, srl, sll`**
- Instruction fields
    - opcode: operation code
    - rs: first source register number
    - rt: second source register number
    - rd: destination register number
    - shamt: shift amount
    - funct: function code (extends opcode)

R-type – Instruction Example

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

`subu $v1, $zero, $t0`

Order of operands: `subu rd, rs, rt`

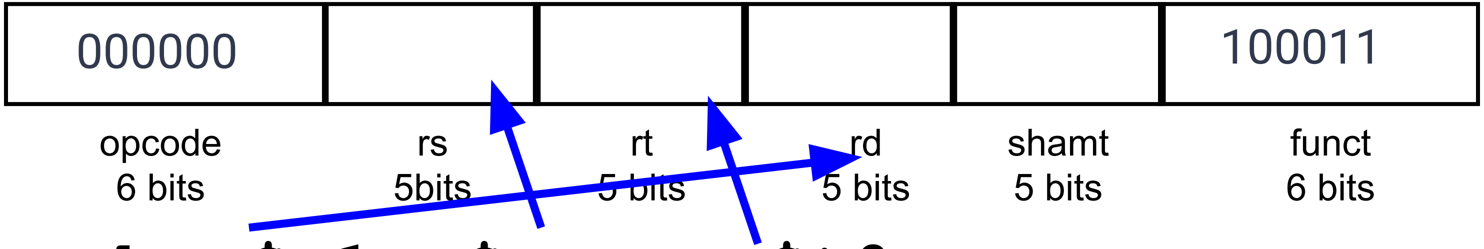


CORE INSTRUCTION SET

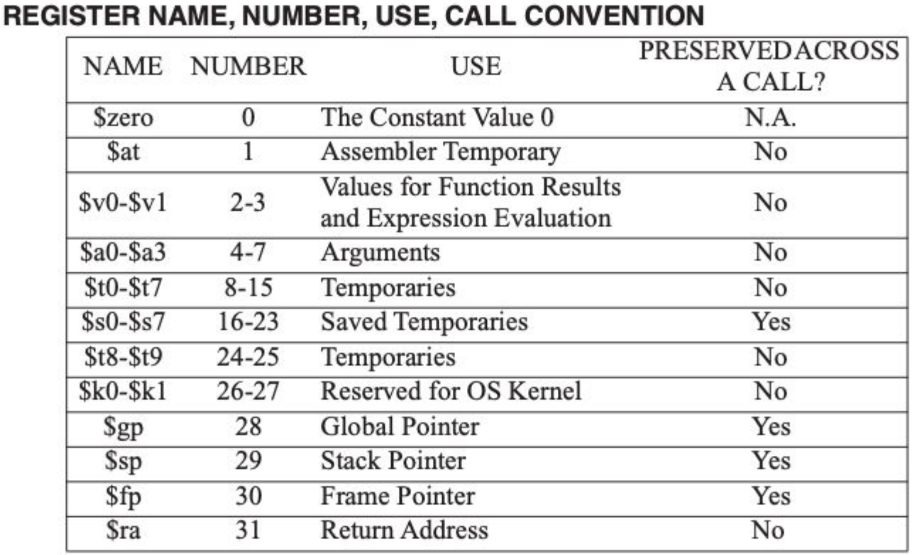
NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

All R-type instructions have opcode = 0

R-type – Instruction Example



```
subu $v1, $zero, $t0
```



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	No

Use the table for register mapping

$v1 = ______ (0b_______)
$zero = ______ (0b_______)
$t0 = ______ (0b_______)

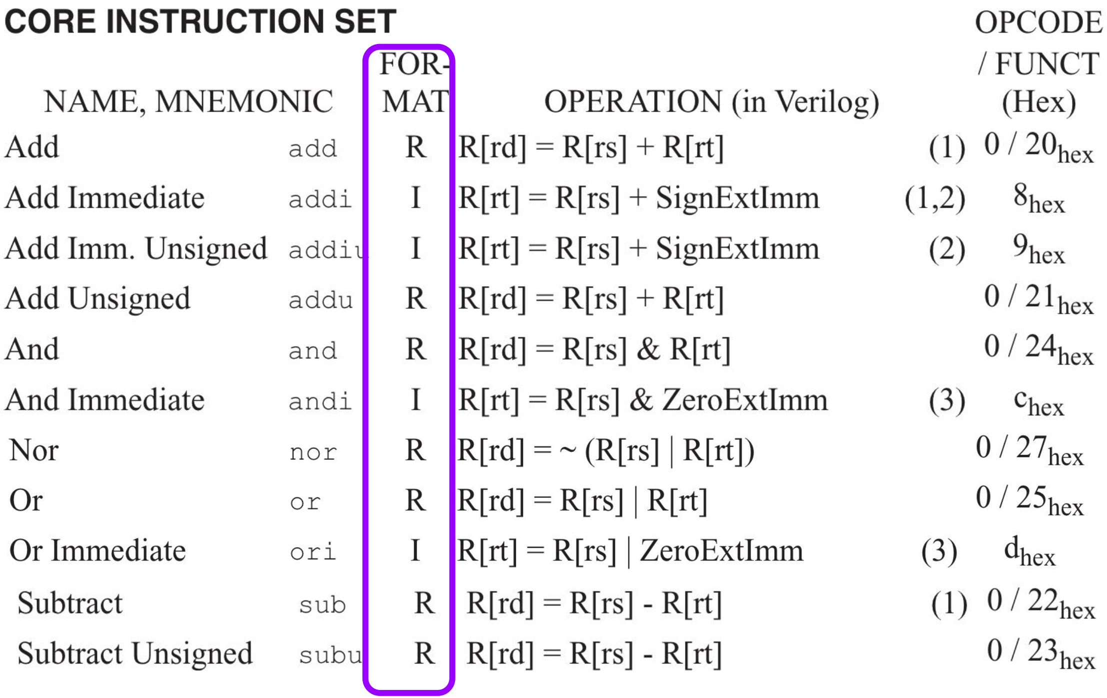
R-type – Instruction Example

Instruction in Binary	000000	00000	01000	00011	00000	100011
	opcode 6 bits	rs 5bits	rt 5 bits	rd 5 bits	shamt 5 bits	funct 6 bits

Convert to Hex

`subu $v1, $zero, $t0` → ____________$_{16}$ in machine language

I–type instructions

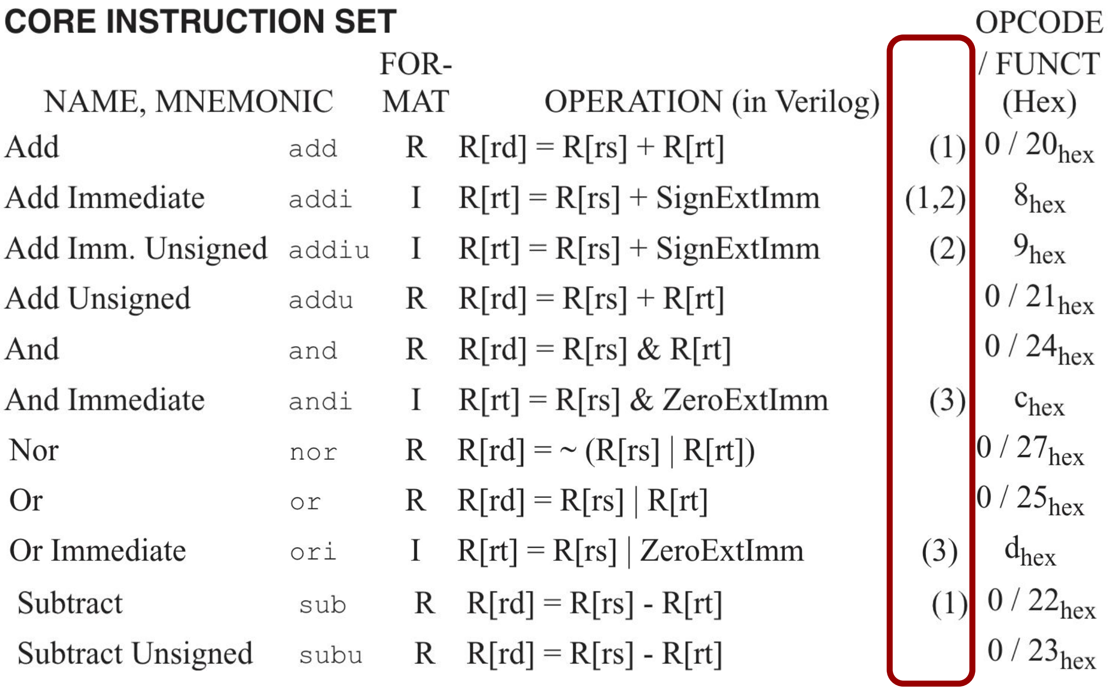


CORE INSTRUCTION SET

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}
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}
Subtract	sub	R	R[rd] = R[rs] - R[rt]	(1)	$0 / 22_{hex}$
Subtract Unsigned	subu	R	R[rd] = R[rs] - R[rt]		$0 / 23_{hex}$

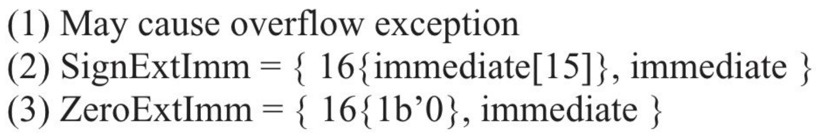
Next let's study I-type instructions

I-type instructions



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}
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}
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 }

In this column are footnotes

- (#) are some instruction specific notes
- What does SignExtImm and ZeroExtImm mean?

Example:

```
addi and addiu verilog
has SignExtImm
```

Review: Sign Extension Immediate

- Sign bit copied to MSB's
- Number value is same

- **Example 1:**
  - 16-bit representation of 3 = 0000_0000_0000_0011
  - 32-bit sign-extended value: 0000_0000_0000_0000_0000_0000_0000_0011 is still 3
- **Example 2:**
  - 16-bit representation of -7 = 1111_1111_1111_1001
  - 32-bit sign-extended value: 1111_1111_1111_1111_1111_1111_111_1001 is still -7

Zero Extension Immediate

- Zeros copied to MSB's
- Value changes for negative numbers

- **Example 1:**
  - 16-bit representation of 3 = 0000_0000_0000_0011
  - 32-bit sign-extended value: 0000_0000_0000_0000_0000_0000_0000_0011 is still 3
- **Example 2:**
  - 16-bit representation of -7 = 1111_1111_1111_1001
  - 32-bit sign-extended value: 0000_0000_0000_0000_1111_1111_111_1001 is not -7

I-type Instructions

opcode	rs	rt	constant or address offset
6 bits	5 bits	5 bits	16 bits

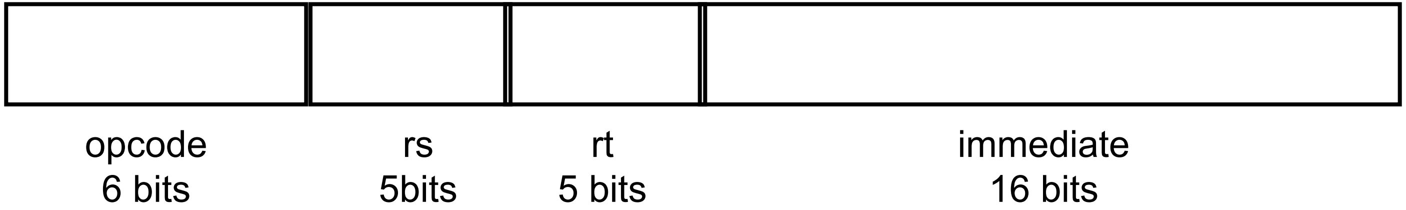
- For immediate arithmetic and Load/Store instructions
  - **opcode:** operation code (unsigned)
  - **rs:** source register number (unsigned)
  - **rt:** destination or source register number (unsigned)
  - **constant:** two's complement constant

    OR
  - **address offset:** two's complement constant added to base address

e.g. addi, andi, ori

e.g. lw/sw

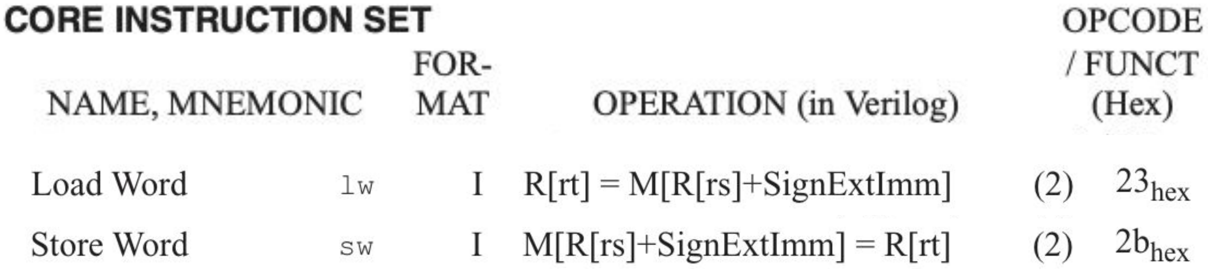
I-type Instruction Example – My Turn



opcode 6 bits	rs 5bits	rt 5 bits	immediate 16 bits

```
lw $t4, 8($s3)
```

Order of operands: `lw rt, Imm(rs)`

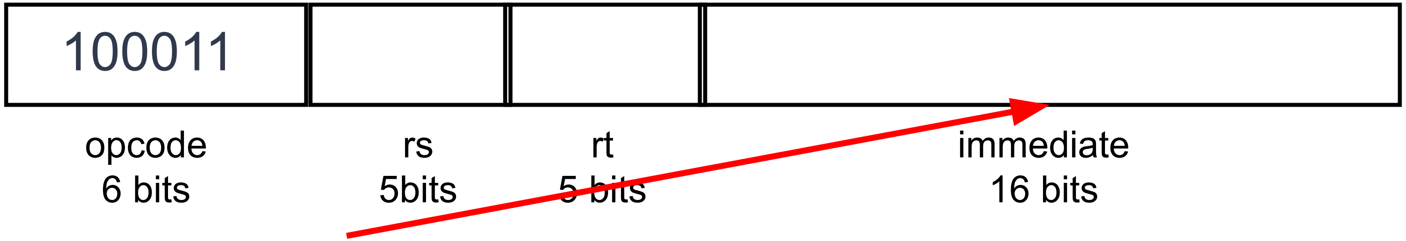


CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Load Word	`lw`	I	R[rt] = M[R[rs]+SignExtImm]	(2)	23_{hex}
Store Word	`sw`	I	M[R[rs]+SignExtImm] = R[rt]	(2)	$2b_{hex}$

If there is just one number then it is only opcode, no funct

I-type Instruction Example – My Turn



100011

opcode
6 bits

rs
5bits

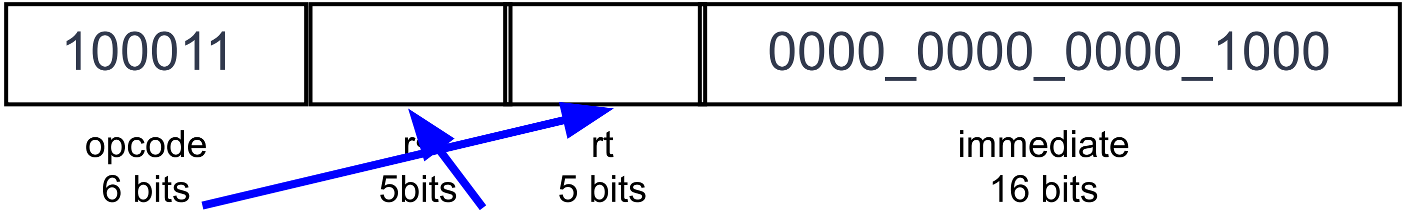
rt
5 bits

immediate
16 bits

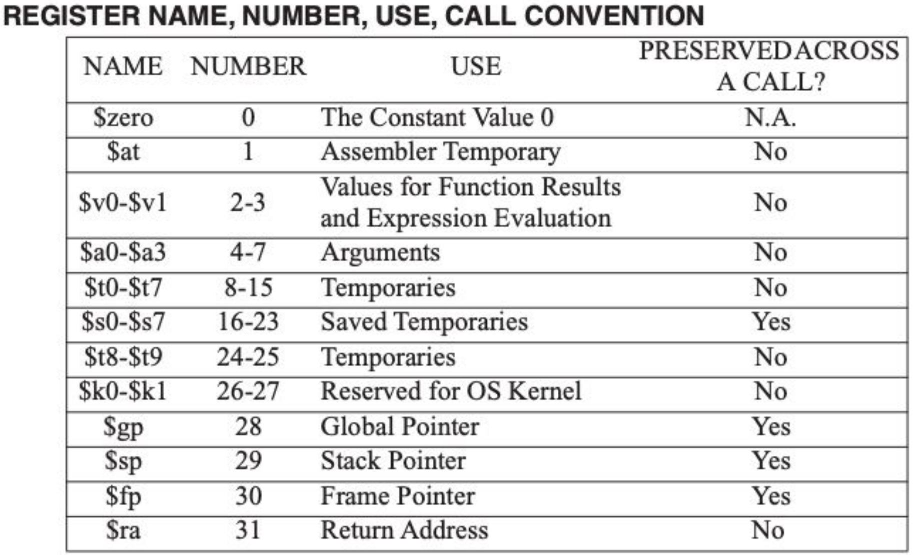
```
lw $t4, 8($s3)
```

Order of operands: `lw rt, Imm(rs)`

I-type Instruction Example – My Turn



```
lw $t4, 8($s3)
```



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	No

Use the table for register mapping

$t4 = ______ (0b_______)

$s3 = ______ (0b_______)

I-type Instruction Example – My Turn

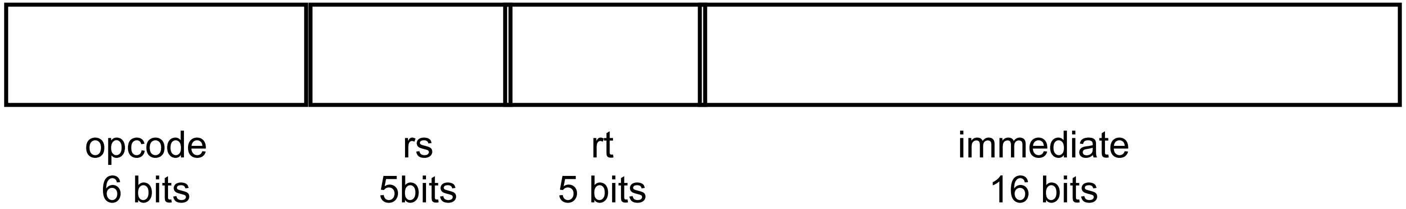
Instruction in Binary

100011	10011	01100	0000_0000_0000_1000
opcode 6 bits	rs 5bits	rt 5 bits	immediate 16 bits

Convert to Hex

`lw $t4, 8($s3)` → ______________$_{16}$ in machine language

I-type Instruction Example – Your Turn



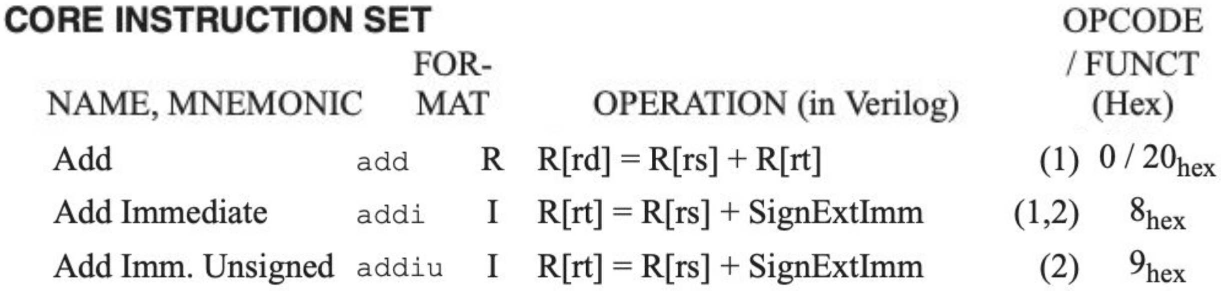
opcode
6 bits

rs
5bits

rt
5 bits

immediate
16 bits

```
addi $t0, $s1, 5
```



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}

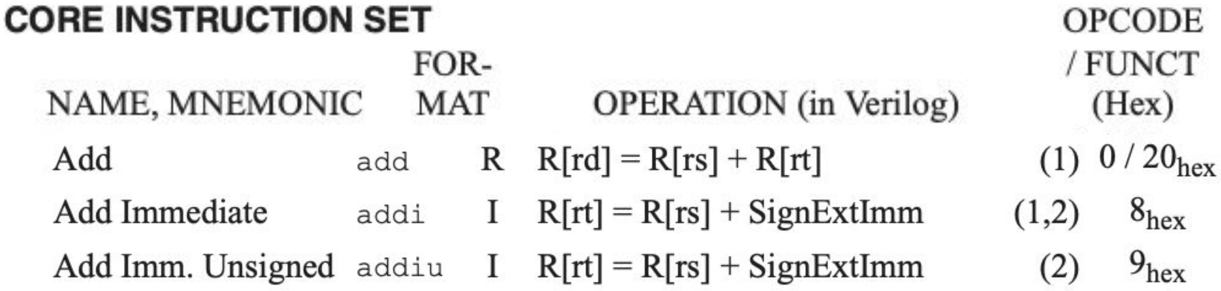
If there is just one number then it is only opcode, no funct

I-type Instruction Example – Your Turn

opcode 6 bits	rs 5bits	rt 5 bits	immediate 16 bits

```
addi $t0, $s1, 5
```

Order of operands: `addi rt, rs, Imm`

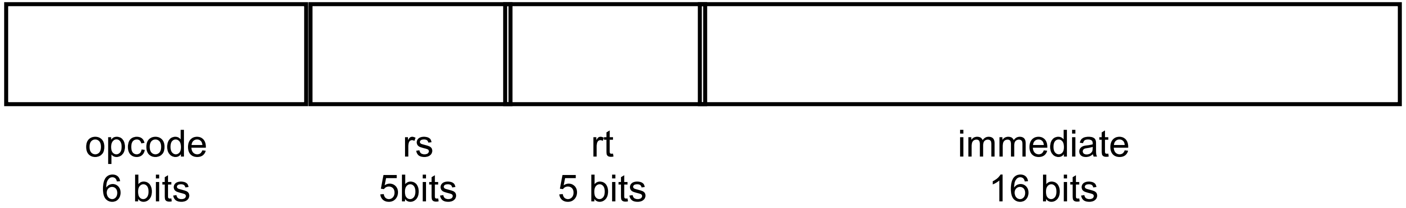


CORE INSTRUCTION SET

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}

If there is just one number then it is only opcode, no funct

I-type Instruction Example – Your Turn



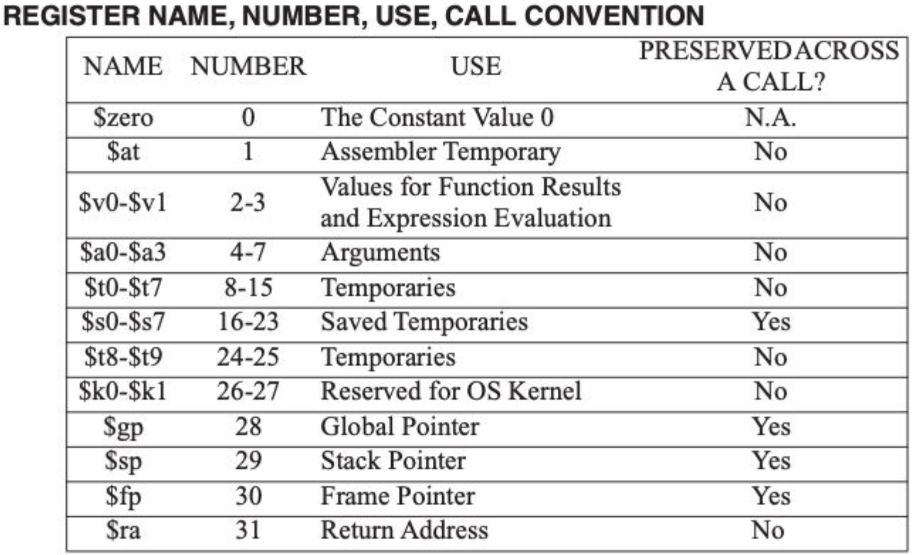
opcode
6 bits

rs
5bits

rt
5 bits

immediate
16 bits

```
addi $t0, $s1, 5
```



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	No

Use the table for register mapping
$t0 = ______ (0b________)
$s1 = ______ (0b________)

I-type Instruction Example – Your Turn

Instruction in Binary

opcode 6 bits	rs 5bits	rt 5 bits	immediate 16 bits

Convert to Hex

`addi $t0, $s1, 5` → ____________________$_{16}$ in machine language

Machine code back to instruction – My Turn

Convert R-type 0x0107482a to MIPS assembly code

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

Convert to binary 0b____

opcode =

[Note all R-type have opcode = 0]

funct =

Ans: ______________________

rs =

rt =

rd =

Review: Branch and Jump

- **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

Allows us to perform if, while and for loops

I–type Branch instruction

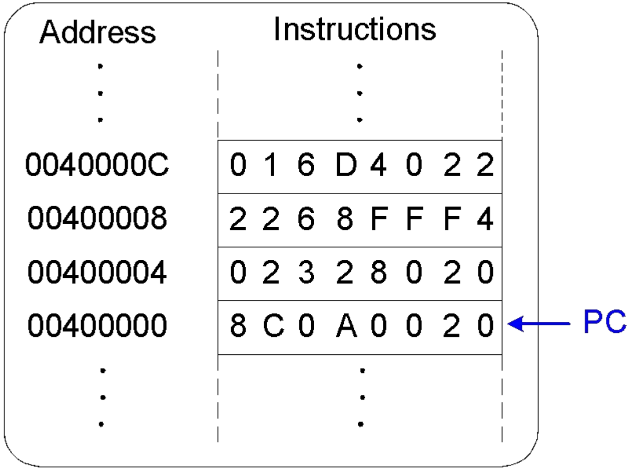
opcode	rs	rt	address offset
6 bits	5 bits	5 bits	16 bits

- Branch instructions specifies
  - Opcode, two registers, 16-bit address offset
  - Note, target address is 32-bit
- Most branch targets are near branch in the memory
  - The offset is relative to the PC, that’s why…
- Branch is also PC-relative addressing
  - Target address = (PC+4) + offset × 4
  - Note, PC is already incremented by 4

Review: Program Counter

Assembly Code	Machine Code
`lw   $t2, 32($0)`	0x8C0A0020
`add  $s0, $s1, $s2`	0x02328020
`addi $t0, $s3, -12`	0x2268FFF4
`sub  $t0, $t3, $t5`	0x016D4022

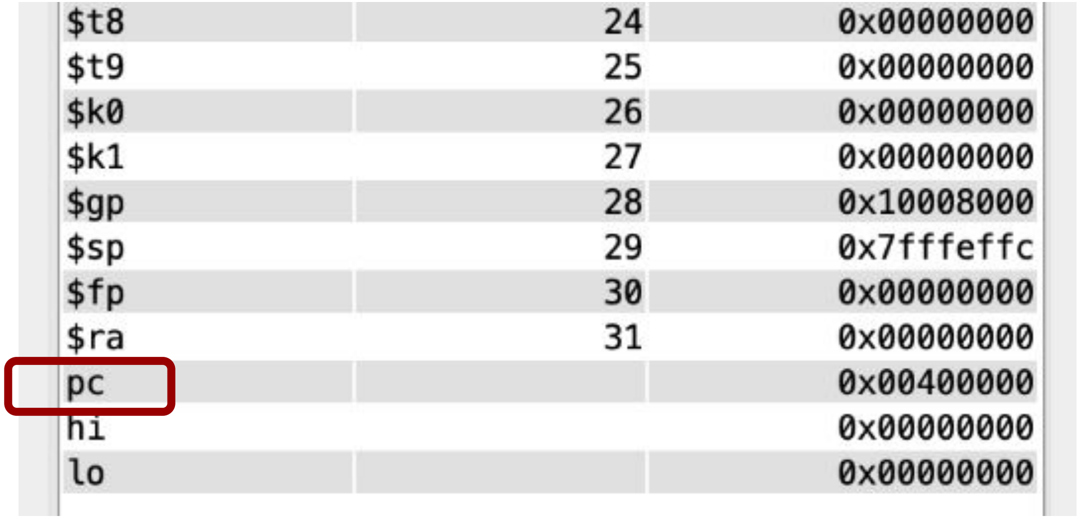
Stored Program



Address	Instructions
⋮	⋮
0040000C	0 1 6 D 4 0 2 2
00400008	2 2 6 8 F F F 4
00400004	0 2 3 2 8 0 2 0
00400000	8 C 0 A 0 0 2 0
⋮	⋮

Main Memory

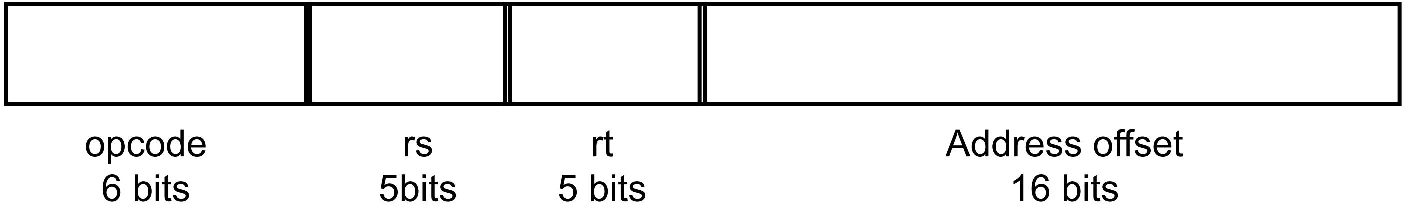
- Program counter (PC) is a 32-bit register that contains the address of the current instruction being executed



$t8	24	0x00000000
$t9	25	0x00000000
$k0	26	0x00000000
$k1	27	0x00000000
$gp	28	0x10008000
$sp	29	0x7fffeffc
$fp	30	0x00000000
$ra	31	0x00000000
pc		0x00400000
hi		0x00000000
lo		0x00000000

PC shown in MARS

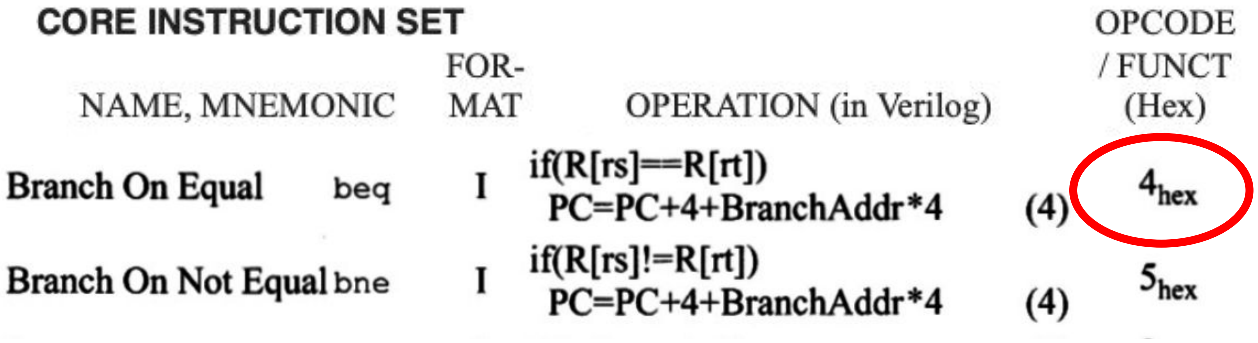
I-type Branch – My Turn



opcode	rs	rt	Address offset
6 bits	5bits	5 bits	16 bits

```
beq $t0, $t1, IfEqualCode
```

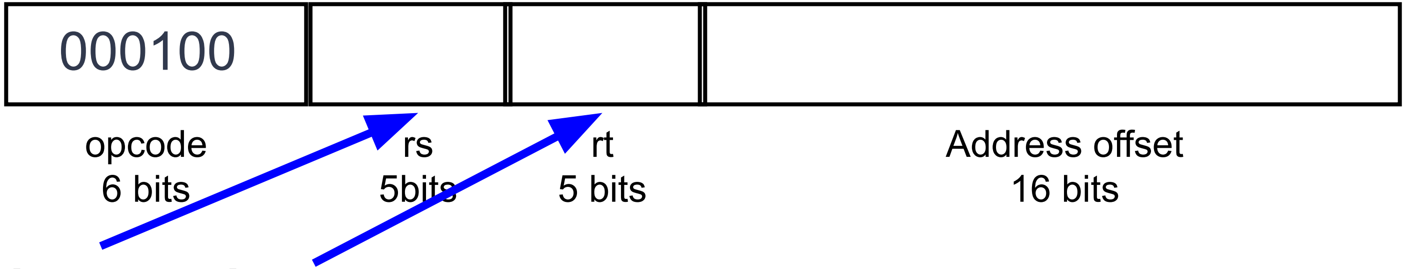
Order of operands: `beq rs, rt, Label`



CORE INSTRUCTION SET

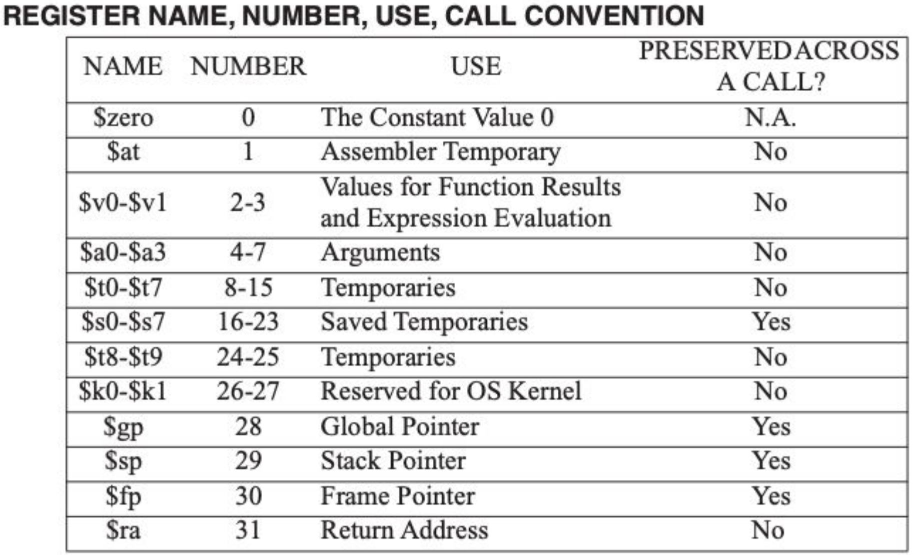
NAME, MNEMONIC		FOR- MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Branch On Equal	beq	I	if(R[rs]==R[rt]) PC=PC+4+BranchAddr*4	(4)	4_{hex}
Branch On Not Equal	bne	I	if(R[rs]!=R[rt]) PC=PC+4+BranchAddr*4	(4)	5_{hex}

I-type Branch – My Turn



000100			
opcode 6 bits	rs 5bits	rt 5 bits	Address offset 16 bits

```
beq $t0, $t1, IfEqualCode
```



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	No

Use the table for register mapping

$t0 = ______ (0b_______)

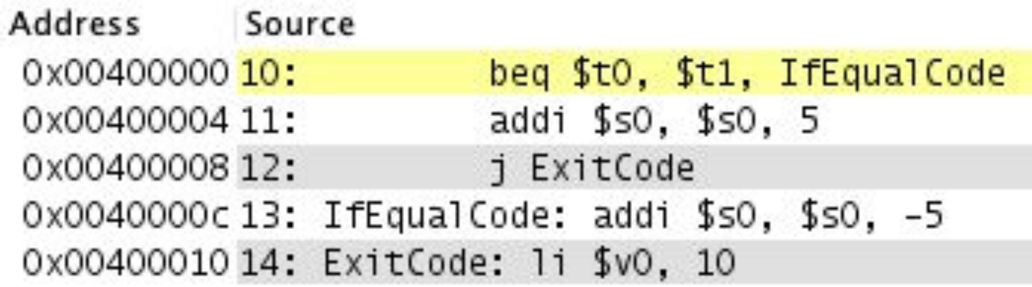
$11 = ______ (0b_______)

I-type Branch – My Turn

000100	01000	01001	
opcode 6 bits	rs 5bits	rt 5 bits	Address offset 16 bits

```
beq $t0, $t1, IfEqualCode
```

Calculate the Address offset by taking the difference between the Branch Target Address (IfEqualCode) vs the Current Instruction (PC+4)



	Address	Source
PC	0x00400000	10: beq $t0, $t1, IfEqualCode
PC+4	0x00400004	11: addi $s0, $s0, 5
PC+8	0x00400008	12: j ExitCode
PC+12	0x0040000c	13: IfEqualCode: addi $s0, $s0, -5
PC+16	0x00400010	14: ExitCode: li $v0, 10

Offset = 2 instructions
(IfEqualCode vs PC+4)

I-type Branch – My Turn

Instruction in Binary

000100	01000	01001	0000_0000_0000_0010
opcode 6 bits	rs 5bits	rt 5 bits	Address offset 16 bits

Convert to Hex

`beq $t0,$t1,IfEqualCode` → ____________$_{16}$ in machine language

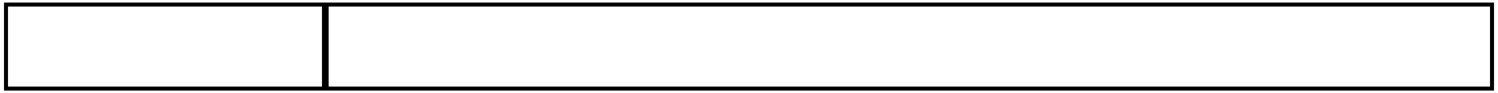
Note: If a branch backwards, then the offset is negative

J-type Instructions - PC-relative addressing

opcode	address
6 bits	26 bits

- All instructions that jump to a target address in the .text segment
  - **`j`** (jump), **`jal`** (jump-and-link) *in a future lecture*
- 2 Instruction fields
  - **opcode:** operation code
  - **address:** Encode the (almost) full address into the instruction
    - Almost because a full address is 32-bits. We will borrow bits from the PC
- PC-relative addressing
    - Jump Target Address = {PC[31:28], address × 4}
    - Note: PC is already incremented by 4

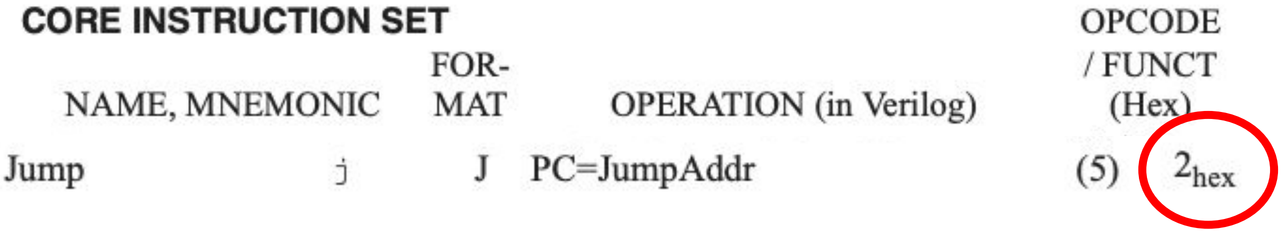
J-type Instruction – My Turn



opcode
6 bits

Address offset
26 bits

`j ExitCode`



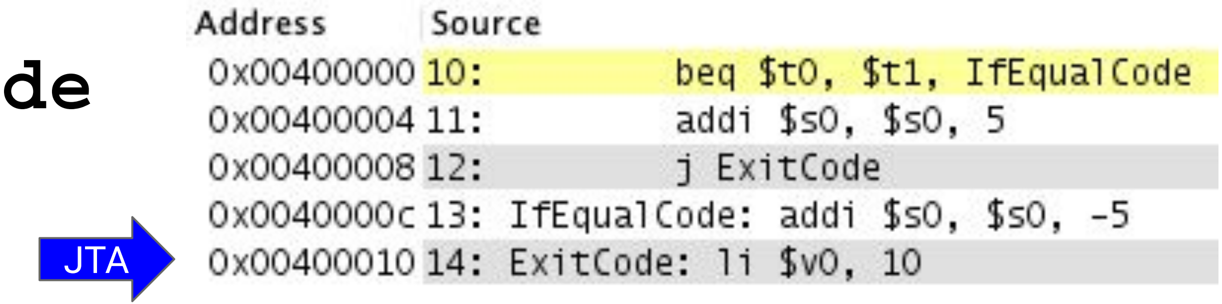
CORE INSTRUCTION SET

NAME, MNEMONIC		FOR-MAT	OPERATION (in Verilog)		OPCODE / FUNCT (Hex)
Jump	j	J	PC=JumpAddr	(5)	2_{hex}

J-type Instruction – My Turn

000010	
opcode 6 bits	Address offset 26 bits

`j ExitCode`



```
Address     Source
0x00400000 10:              beq $t0, $t1, IfEqualCode
0x00400004 11:              addi $s0, $s0, 5
0x00400008 12:              j ExitCode
0x0040000c 13: IfEqualCode: addi $s0, $s0, -5
0x00400010 14: ExitCode: li $v0, 10
```

Compute 26-bit address offset from a 32-bit Label in 4 steps

(1) Start with Label (ExitCode) or Jump Target Address (JTA) ____________________

(2) Remove top 4 bits (equivalent to mod 28) ____________________

(3) Shift right by 2 (equivalent to divide by 4) ____________________

(4) Address is the resulting 26-bits ____________________

J-type Instruction – My Turn

Instruction in Binary

000010	00_0001_0000_0000_0000_0000_0100
opcode 6 bits	Address offset 26 bits

Convert to Hex

`j ExitCode` → ____________$_{16}$ in machine language

Summary

- MIPS has 3 instruction formats
  - R-, I- and J- type
- Practiced being an Assembler and converted assembly code to machine code and back

Next

Leaf and Non-leaf procedures