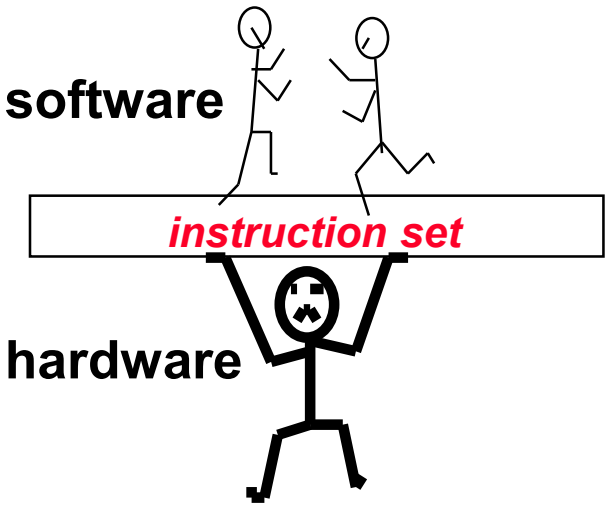
Chapter 2

Instructions：Language of the Computer



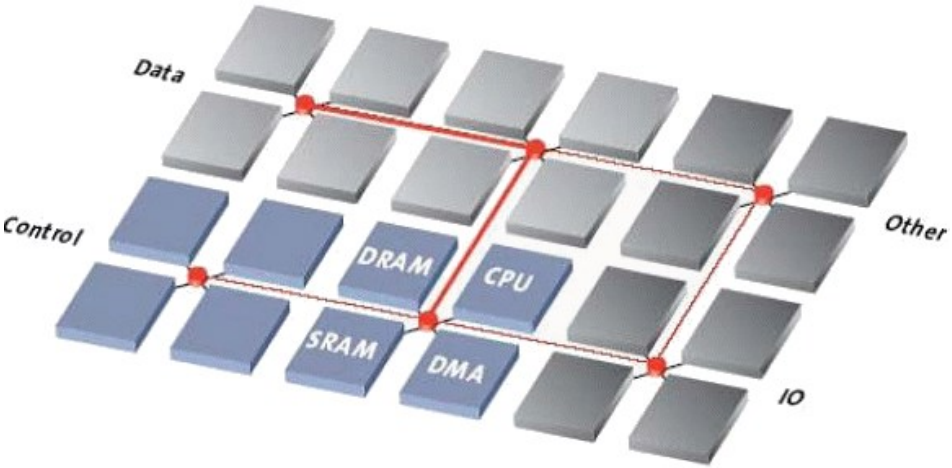
Kuei-Chung Chang

Email: changkc@fcu.edu.tw

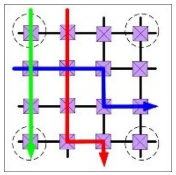
Ext. 3753, Office 211

ation Engineering and Computer Science

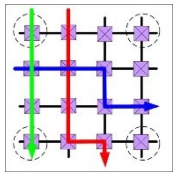
Feng Chia Univ.



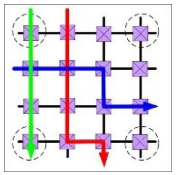
Spring 2019



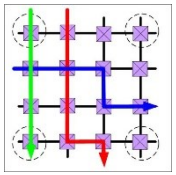
Outline



Outline

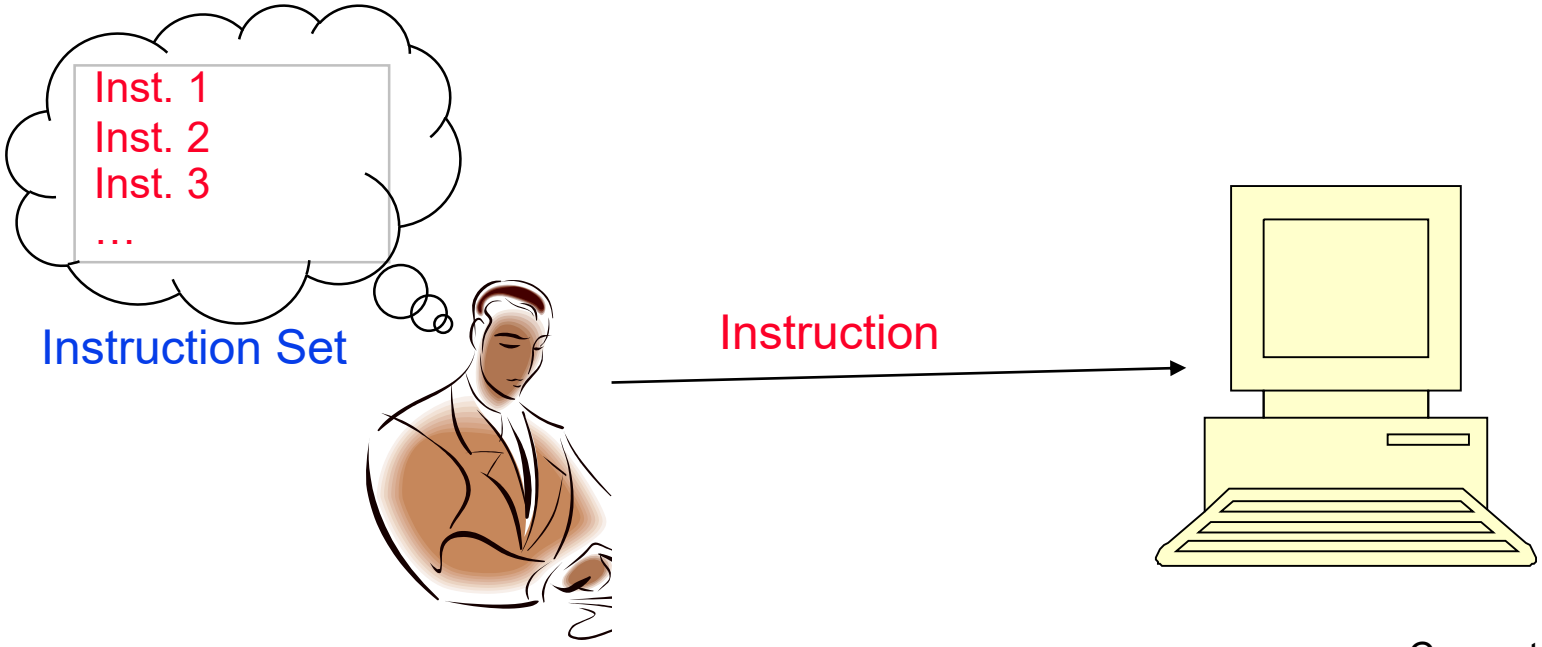


Outline



Introduction

- What is Instruction ?
  - Language of the Machine
- The words of a computer's language are called instructions, and its vocabulary is called an instruction set.



Instructions are Like Car Pedals

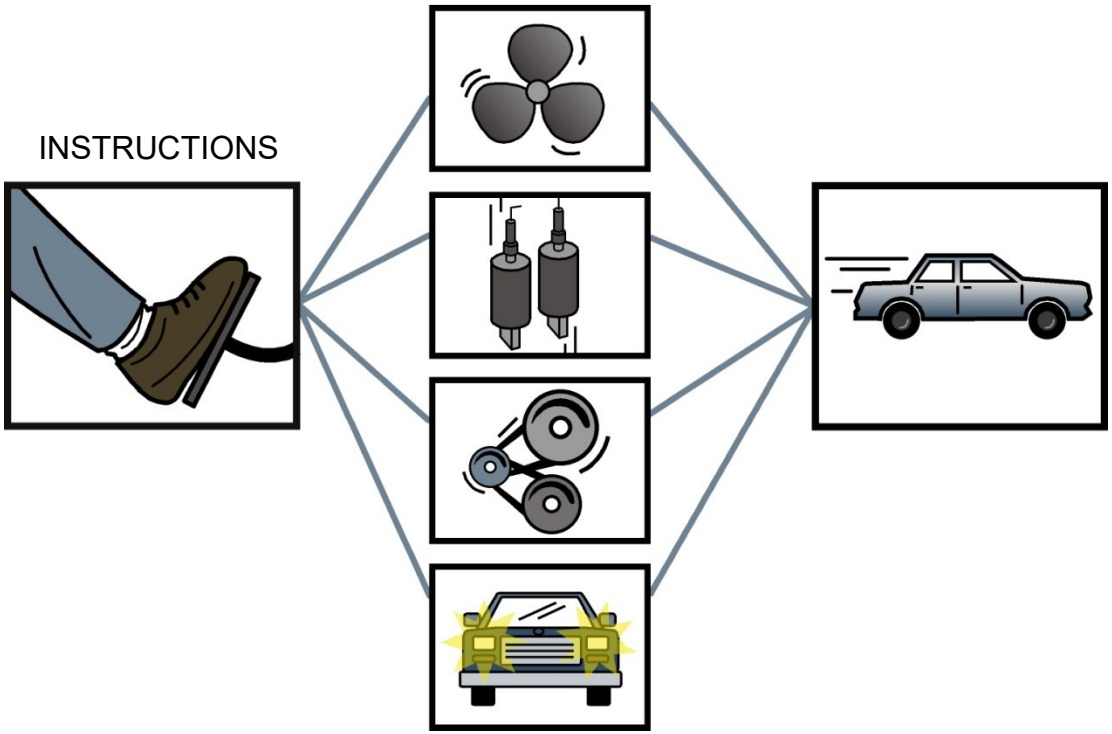
a[i] = b[i] + c;

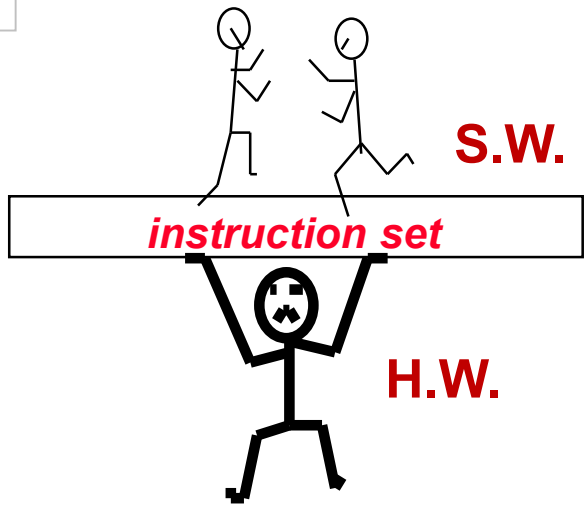
```
lw    $15, 0($2)
add   $16, $15, $14
```

```
0000001011000000
1101000001000010
```

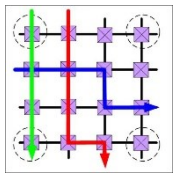
MICROARCHITECTURE
("UNDER THE HOOD")

INSTRUCTIONS





See Patt & Patel text, Andelman Decl. in support of ARM's Motion for Summary Judgment, Ex. C, at 12.



ISA Example: RISC vs. CISC

RISC (Reduced Instruction Set Computer)

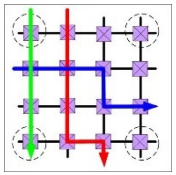
Concept: Keep operations simple to allow fast clock

1. **Load-store architecture**
   `add $t0, $s2, $t0`
2. **Fixed-width instruction encoding**
3. **Large register files**
4. **Big program code size**
5. **Complex compiler**

CISC (Complex Instruction Set Computer)

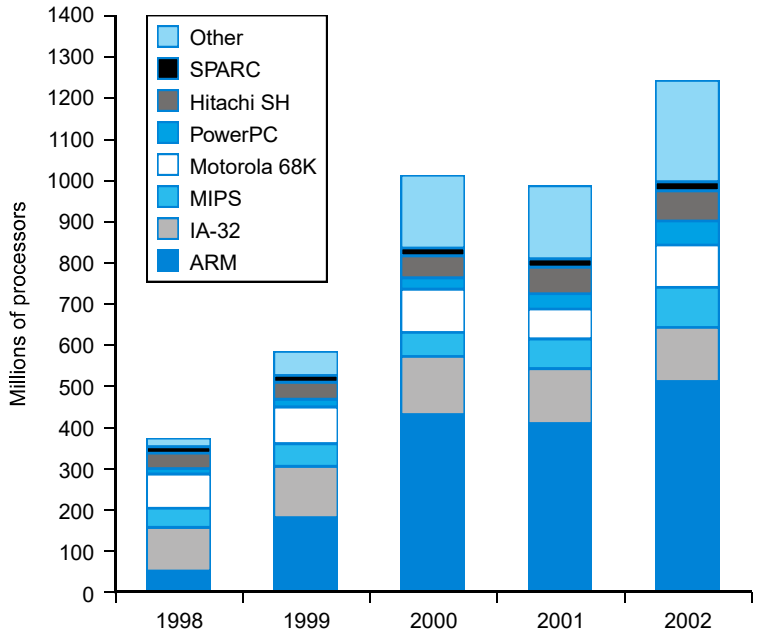
Concept : Implement complex instructions to reduce # instructions in programs

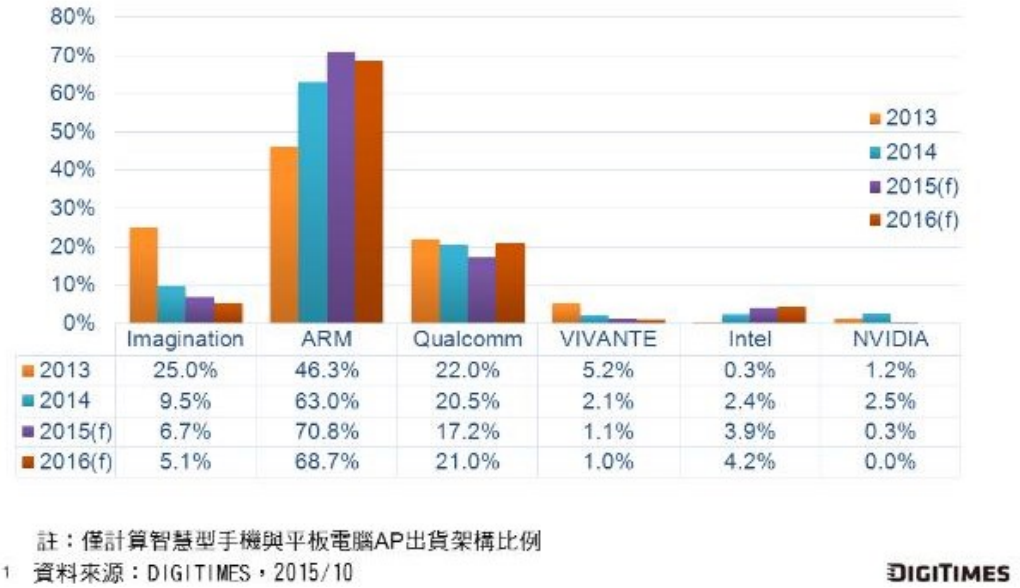
1. Arithmetic instructions access memory `add 12($s3), $s2, 8($s3)`
2. Variable-width instruction encoding
3. Small register files
4. Small program code size
5. Complex HW/Micro program



The focused ISA: ***MIPS***

- We'll be working with the MIPS instruction set architecture
  - similar to other architectures developed since the 1980's
  - Almost 100 million MIPS processors manufactured in 2002
  - used by NEC, Nintendo, Cisco, Silicon Graphics, Sony, …



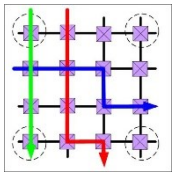


	Imagination	ARM	Qualcomm	VIVANTE	Intel	NVIDIA
2013	25.0%	46.3%	22.0%	5.2%	0.3%	1.2%
2014	9.5%	63.0%	20.5%	2.1%	2.4%	2.5%
2015(f)	6.7%	70.8%	17.2%	1.1%	3.9%	0.3%
2016(f)	5.1%	68.7%	21.0%	1.0%	4.2%	0.0%

註：僅計算智慧型手機與平板電腦AP出貨架構比例

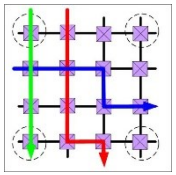
資料來源：DIGITIMES，2015/10

DIGITIMES

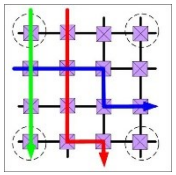


MIPS - A "Typical" RISC

- A RISC architecture
- 32-bit byte addresses
- Load/Store - only displacement
- Registers
  - 32 32-bit General-Purpose Registers
- Single address mode for load/store: base + displacement
  - no indirection
- ***Emphasize***
  - A simple load/store instruction set
  - Design for pipelining efficiency
  - An easily decoded instruction set
  - Efficiency as a compiler target



Outline



MIPS arithmetic

- All instructions have 3 operands
- Operand order is fixed (*destination first*)

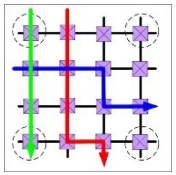
Example:

C code: `a = b + c`

MIPS code: `add a, b, c`

(we'll talk about registers in a bit)

"The natural number of operands for an operation like addition is three…requiring every instruction to have exactly three operands, no more and no less, conforms to the philosophy of keeping the hardware simple"



MIPS arithmetic

- Operands must be registers, only 32 registers provided
- Each register contains 32 bits
- *Of course this complicates some things...*

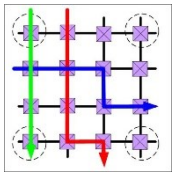
C code:

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

MIPS code:

```
add a, b, c
add a, a, d
```

- It takes 2 instructions to complete the arithmetic



MIPS arithmetic

- Complex C program

C code **f = (g+h) - (i+j)**

MIPS code

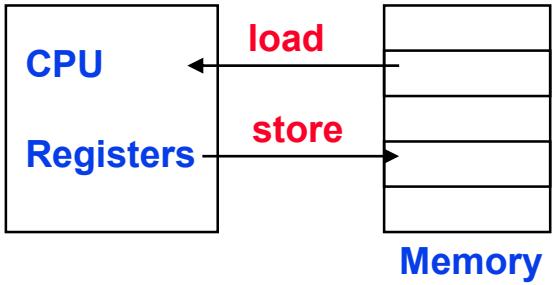
```
add  t0,  g,  h
add  t1,  i,  j
sub  f  ,  t0,  t1
```

- **Design Principle 1**: simplicity favors regularity.

HW for a variable number of operands is more complicated than HW for a fixed number.

Lw/Sw Instruction example – Byte unit

- Load (lw) and store (sw) instructions
- Example:

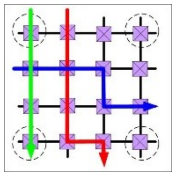


C code: `A[12] = h + A[8];`

MIPS code:

```
lw    $t0, 8($s3)
add   $t0, $s2, $t0
sw    $t0, 12($s3)
```

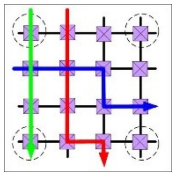
- Can refer to registers by name (e.g., $s2, $t2) instead of number
- Variable **h** is associated with register $s2, and the base address of the **Array A** is in $s3
- Store word has destination last
- *Remember arithmetic operands are registers, not memory!*
- Can't write: `add 12($s3), $s2, 8($s3)`



Variables vs. Registers

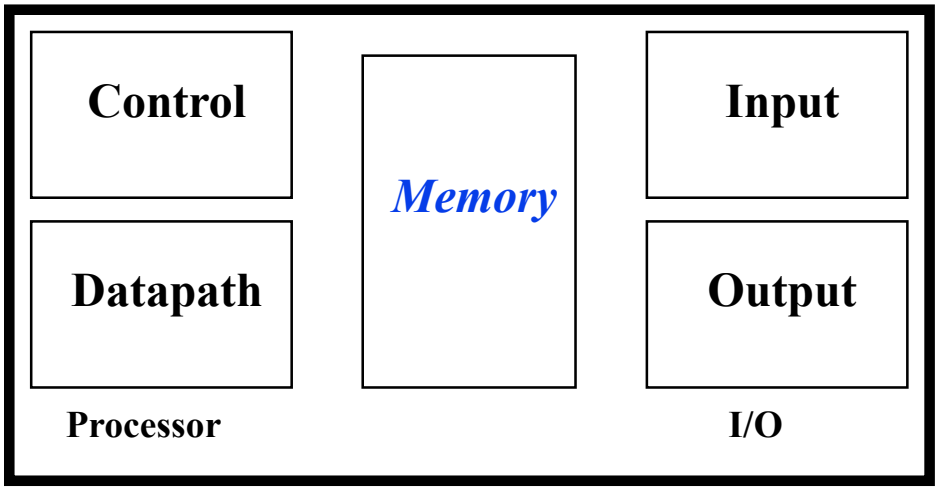
- ***Registers*** ***are primitives used in hardware design***
- The size of a register in the ARM is 32 bits (word).
- The three operands of ARM arithmetic instructions must each be chosen from one of the 16 registers.
- One major difference between the variables of a programming language and registers is the limited number of registers.
- **Design Principle2** : *smaller is faster*.

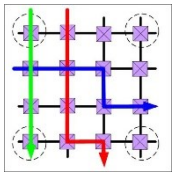
A very large number of registers may increase the clock cycle time because it takes electronic signals longer when they must travel farther.



Registers v.s. Memory

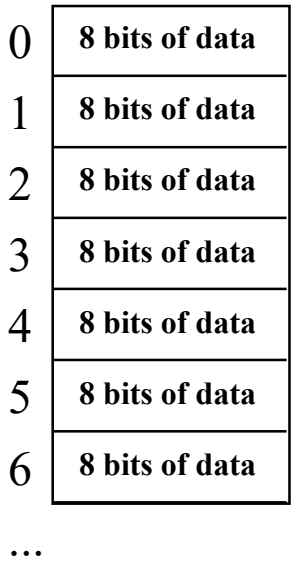
- Arithmetic instructions operands must be registers,
  — only 16 registers provided
- Compiler associates variables with registers
- *What about programs with lots of variables?*
  - Memory

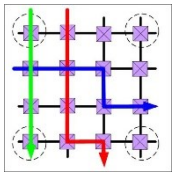




Memory Organization

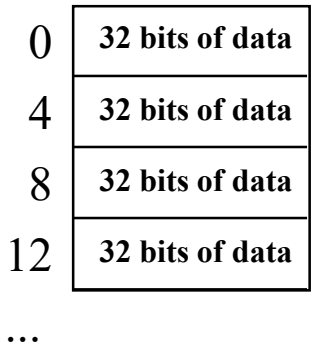
- Viewed as a large, single-dimension ***array***, with an address.
- A memory address is an ***index*** into the array
- "Byte addressing" means that the index points to a byte of memory.





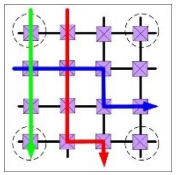
Memory Organization

- Bytes are nice, but most data items use larger "words"
- For ARM, a word is 32 bits or 4 bytes.



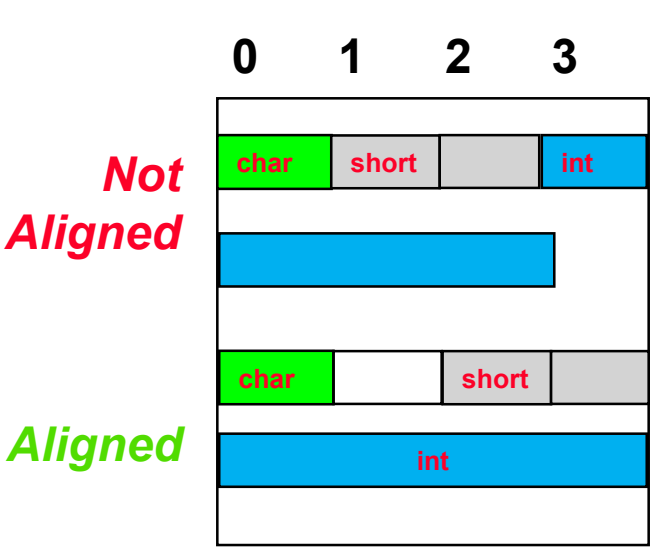
Registers hold 32 bits of data

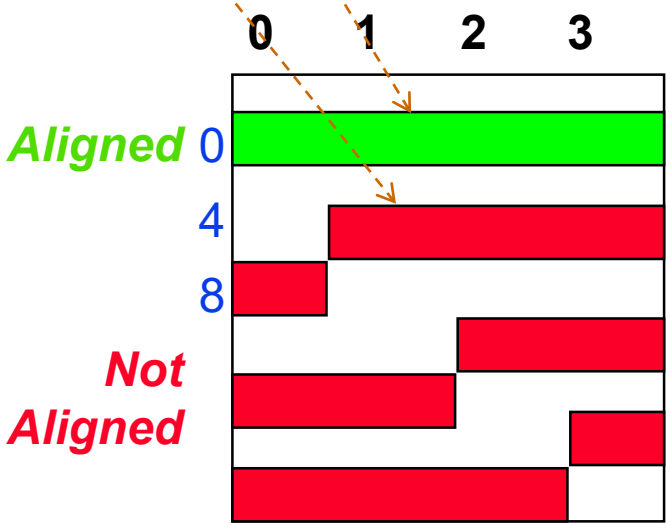
- 2^{32} bytes with byte addresses from 0 to $2^{32}-1$
- 2^{30} words with byte addresses 0, 4, 8, ... $2^{32}-4$

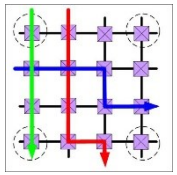


Alignment restriction

- Alignment restriction
  - In MIPS, words must start at addresses that are multiples of 4
  - **Aligned vs. Misaligned (not Aligned)**
    - Variable A allocated in 0-3 (**Aligned**)
      - We can read A in 1-step memory access
    - Variable B allocated in 5-8 (**Misaligned**)
      - We have to take 2 steps to read B







Alignment restriction

- Addressing Operands (Due to "*byte addressing*" …)
  - Endian Convention

    Ordering the bytes within a word

  - **Big Endian**: MSB at xx00 (**MIPS**)

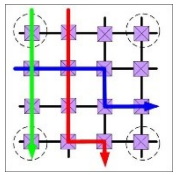
word addr	MSB			LSB
0	0	1	2	3
4	4	5	6	7

ex. : IBM, Motorola

  - **Little Endian**: LSB at xx00 (**ARM**)

word addr	MSB			LSB
0	3	2	1	0
4	7	6	5	4

ex. Intel, Dec



Lw/Sw Instruction example – Word unit

- Load and store instructions
- Example:

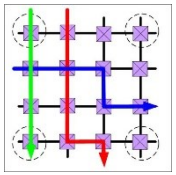
0	32 bits of data
4	32 bits of data
8	32 bits of data
12	32 bits of data

C code: ***A[12] = h + A[8];***

MIPS code:

```
lw    $t0, 32($s3)
add   $t0, $s2, $t0
sw    $t0, 48($s3)
```

- *Remember arithmetic operands are registers, not memory!*
- Can't write: `add 48($s3), $s2, 32($s3)`



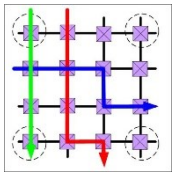
Example

- Can we figure out the code?

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

data swap

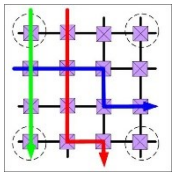
```
swap:
   muli $2, $5, 4
   add $2, $4, $2
   lw $15, 0($2)
   lw $16, 4($2)
   sw $16, 0($2)
   sw $15, 4($2)
   jr $31
```



So far we've learned:

- MIPS arithmetic characteristics
  - — *loading words but addressing bytes*
  - — *arithmetic on registers only*

Instruction	Meaning
`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`
`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`
`lw $s1, 100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1, 100($s2)`	`Memory[$s2+100] = $s1`



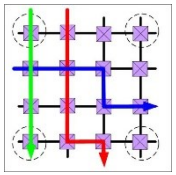
Constant or Immediate Operands

- Many times a program will use a constant in an operation
  - Ex. Index offset
- Example: Use instructions we have learned so far …
  - **index = index + 4; index++; (C code)**

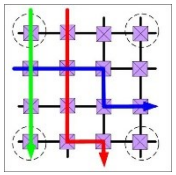
```
• lw    $t0, AddrConstant4($1)  //  $t0 = constant 4
• add   $s3, $s3, $t0           //  $s3 = $s3 + $t0
```

  - *Assuming that AddrConstant4($1) is the memory address of the constant 4*
- An alternative good solution …***Immediate instructions***
  - addi $s3, $s3, ***4*** // $s3 = $s3 + 4
- ***Design Principle 3****: Make the common case fast.*

Constant operands occur frequently, ….

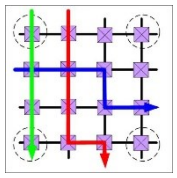


Outline



Numbers

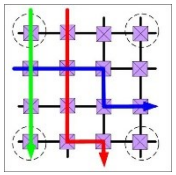
- **Bits are just bits** (no inherent meaning)
  — conventions define relationship between bits and numbers
- ***Binary numbers*** (base 2)
  0000 0001 0010 0011 0100 0101 0110 0111 1000 1001...
  decimal: $0...2^n-1$
- Of course it gets more complicated:
  - *numbers are finite (ex. overflow、underflow)*
  - *fractions and real numbers (ex. 1.8899)*
  - *negative numbers (ex. -1)*
- ***How do we represent negative numbers?***



Possible Representations for negative numbers

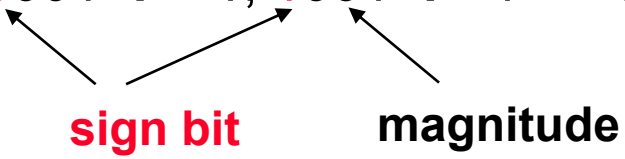
	Sign and Magnitude:	One's Complement	Two's Complement
+	000 = +0	000 = +0	000 = +0
+	001 = +1	001 = +1	001 = +1
+	010 = +2	010 = +2	010 = +2
+	011 = +3	011 = +3	011 = +3
-	100 = -0	100 = -3	100 = -4
-	101 = -1	101 = -2	101 = -3
-	110 = -2	110 = -1	110 = -2
-	111 = -3	111 = -0	111 = -1

- Which one is best? Why?
- Issues:
  - Balance ? (*Problem in 2's complement*)
  - Number of zeros ? (*in Sign Magnitude, 1's complement*)
  - Ease of operations ? (*in Sign Magnitude, 1's complement*)

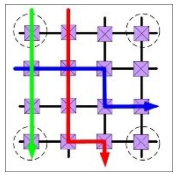


Sign Magnitude

- To add a separate sign bit
  - 0001 → +1, 1001 → -1 ➔ ***how about (-1) + (1) = ?***



- Shortcomings:
  - Where to put the sign bit? Right/Left?
  - Adders need an extra step to set the sign
  - *Two zeros ➔ positive zero and negative zero*
    - *0000 → +0, 1000 → -0*
    - *the same with one's complement*



ARM – 2's complement

- 32 bit signed numbers:

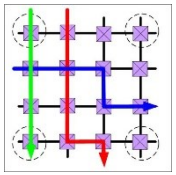
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = +\ 2_{ten}$
...
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$ ***maxint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$ ***minint***
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -\ 2{,}147{,}483{,}646_{ten}$
...
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -\ 3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

Imbalance problem :

The negative number – 2,147,483,648 has no corresponding positive number.

But, sign magnitude had more problems …

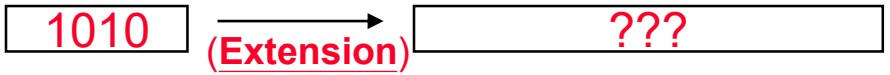
Every computer today uses 2's complement to represent signed numbers.



Two's Complement Operations

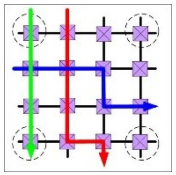
- Negating a two's complement number: **invert all bits and add 1**
  - remember: "negate" and "invert" are quite different!

- "**sign extension**"



  - Converting n bit numbers into numbers with more than n bits:
  - **16-bit** immediate gets converted to **32 bits** for arithmetic
  - copy the most significant bit (the sign bit) into the other bits

```
0010  -> 0000 0010
1010  -> 1111 1010
```

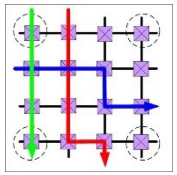


Sign extension - Example

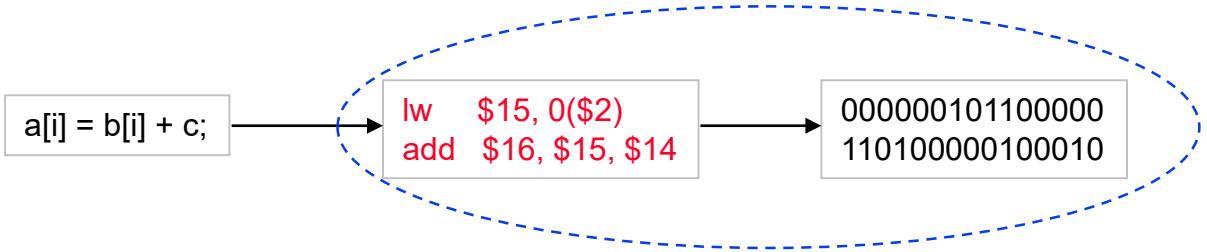
- Convert 16-bit version of 2 and -2 to 32-bit binary numbers.

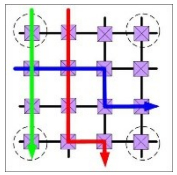
- 2 ➔ **0**000 0000 0000 0010
- 0000 0000 0000 0000 0000 0000 0000 0010

- -2 ➔ **1**111 1111 1111 1110 (2's complement of +2)
- 1111 1111 1111 1111 1111 1111 1111 1110



Outline





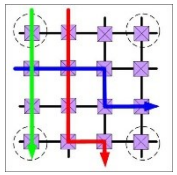
Machine Language

- Instructions, like registers and words of data, are also 32 bits long
  - Example: `add $t1, $s1, $s2`
  - registers have numbers, `$t1=9, $s1=17, $s2=18`

- Instruction Format:

	6 bits	5	5	5	5	6
Format	*op*	rs	rt	**rd**	shamt	*funct*
Example	000000	10001	10010	01001	00000	100000

- *Can you guess what the field names stand for?*
- ***How about load/store instructions?***
  - *(Any problems with this format? addr. boundary < 2^5)*



Machine Language

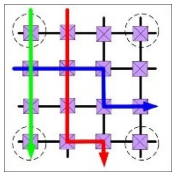
- Consider the load-word and store-word instructions,
  - What would the regularity principle have us do?
  - ***New principle 4: Good design demands a compromise***
- Introduce a new type of instruction format
  - **I-type** for data transfer instructions
  - other format was **R-type** for register
- Example:

```
lw $t0, 32($s2)
```

35	18	9	32
op	rs	rt	*16-bit number*
6 bits	5	5	16 bits

- ***Where's the compromise***?

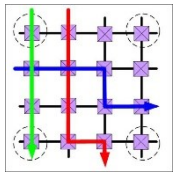
Requiring different kinds of instruction formats …



Summary for I & R type instructions

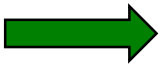
Only one of these columns will be used at the same time

Instruction	Format	op	rs	rt	rd	shamt	funct	address
`add`	R	0	reg	reg	reg	0	32_{10}	n.a
`sub` (subtract)	R	0	reg	reg	reg	0	34_{10}	n.a
`addi` immediate	I	8_{10}	reg	reg	n.a.	n.a.	n.a.	constant
`lw` (load word)	I	35_{10}	reg	reg	n.a.	n.a.	n.a.	address
`sw` (store word)	I	43_{10}	reg	reg	n.a.	n.a.	n.a.	address



Example (p. 84)

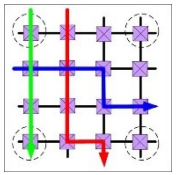
- A[300] = h + A[300];



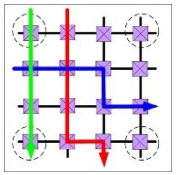
```
lw    $t0, 1200($t1)
add   $t0, $s2, $t0
sw    $t0, 1200($t1)
```

Op	Rs	Rt	Rd	Shamt	funct
35	9	8	1200		
0	18	8	8	0	32
43	9	8	1200		

Op	Rs	Rt	Rd	Shamt	funct
100011	01001	01000	0000 0100 1011 0000		



Outline

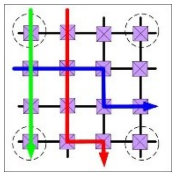


Logical Operations

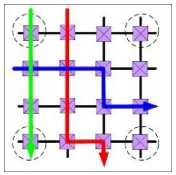
- Instructions for bitwise manipulation

Operation	C	Java	MIPS
Shift left	<<	<<	sll
Shift right	>>	>>>	srl
Bit-by-bit AND	&	&	and, andi
Bit-by-bit OR	|	|	or, ori
Bit-by-bit NOT	~	~	nor

- **sll $t2, $s0, 4 # reg $t2 = reg $s0 << 4 bits**



Outline



Control

- Decision making instructions
  - alter the control flow,
  - i.e., change the "next" instruction to be executed

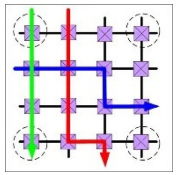
- MIPS ***conditional branch*** instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: `if (i==j) h = i + j;`

```
      bne $s0, $s1, Label
      add $s3, $s0, $s1
Label:  ....
```

- ***Why use opposite test for branch?***



Control

- MIPS ***unconditional branch*** instructions:

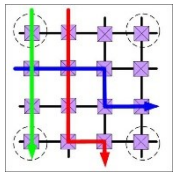
```
j  label
```

- Example:

```
if (i!=j)                beq $s4, $s5, Lab1
    h=i+j;               add $s3, $s4, $s5
else                     j   Lab2
    h=i-j;           Lab1:   sub $s3, $s4, $s5
                     Lab2:   ...
```

- In general, the code will be *more efficient* if we test for the opposite condition to branch over the code that performs the subsequent then part of the if.

- *Can you build a simple for loop?*



Control – P. 92 Example

- Example：while loop

- **while (save[i] == k)**

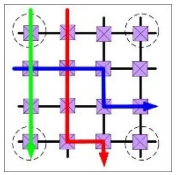
  i += 1;

index i ($s3) = i x 4

base register

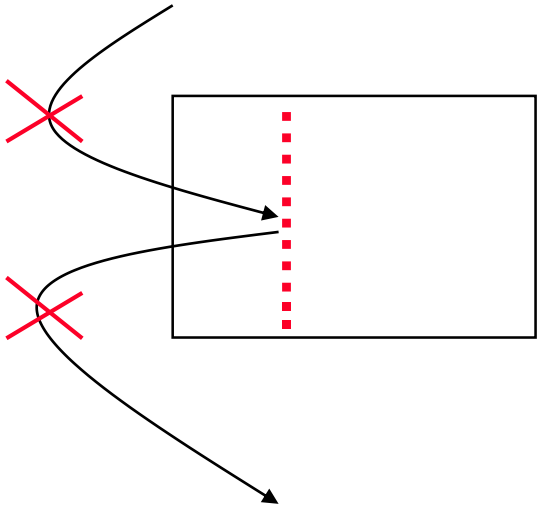
```
Loop:   sll    $t1, $s3, 2     // next address, i*4
        add    $t1, $t1, $s6   // calculate addr. of save[i]
        lw     $t0, 0($t1)     // load the save[i]
        bne    $t0, $s5, Exit  // compare save[i] and k
        addi   $s3, $s3, 1     // i = i + 1
        j      Loop
Exit:
```

k

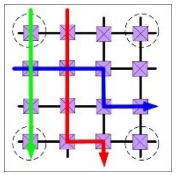


Control

- ***Basic block***
  - A sequence of instructions
    - without *branches*, except possible at the end,
    - and without *branch targets* or *branch labels*, except possibly at the beginning.



```
        inst. 1
        inst. 2
        inst. 3
        bne …
        inst. 4
        inst. 5
        inst. 6
Label: …
        inst. 7
        inst. 8
        inst. 9
        beq
```

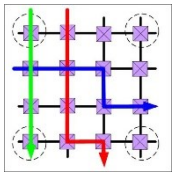


So far:

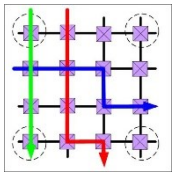
Instruction	Meaning
`add $s1,$s2,$s3`	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	`Memory[$s2+100] = $s1`
`bne $s4,$s5,L`	**Next instr. is at Label if $s4 ≠ $s5**
`beq $s4,$s5,L`	**Next instr. is at Label if $s4 = $s5**
`j Label`	**Next instr. is at Label**

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				



Outline

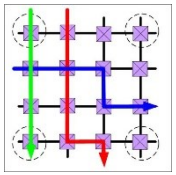


Supporting Procedures

```
void main( ) {
        Int   b = func(p1, p2, p3);
}
```

```
int func(p1, p2, p3 ) {
                ……
                return 0;
}
```

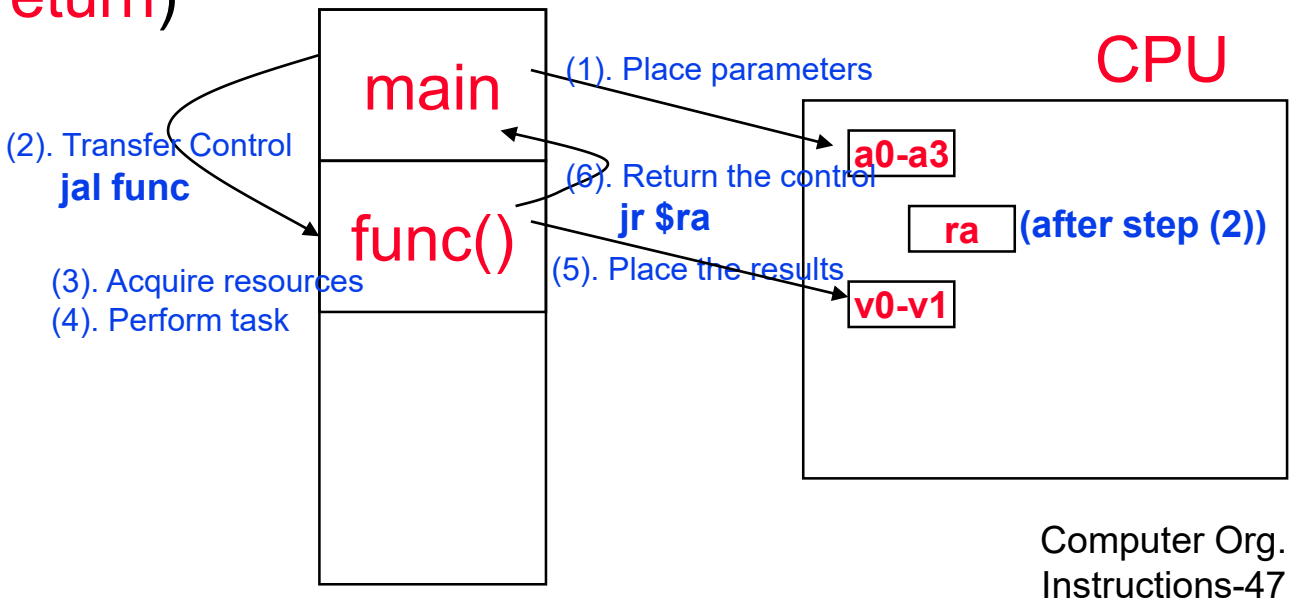
1. Put parameters in a place where the procedure can access them.
2. Transfer control to the procedure
3. Acquire the storage resources needed for the procedure.
4. Perform the procedure task
5. Put the result value in a place where the calling program can access it.
6. Return control to the point of origin

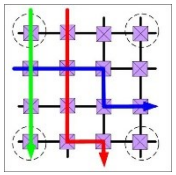


Supporting Procedures

```
void main() {
    • Int b = func(p1, p2, p3);
}
```

- ***Jump-and-link*** (**jal**) :
  It jumps to an address and simultaneously saves the address of the following inst. in $ra (then use **jr – jump register** to return)

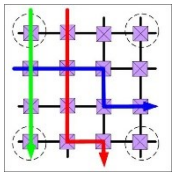




Policy of Use Conventions

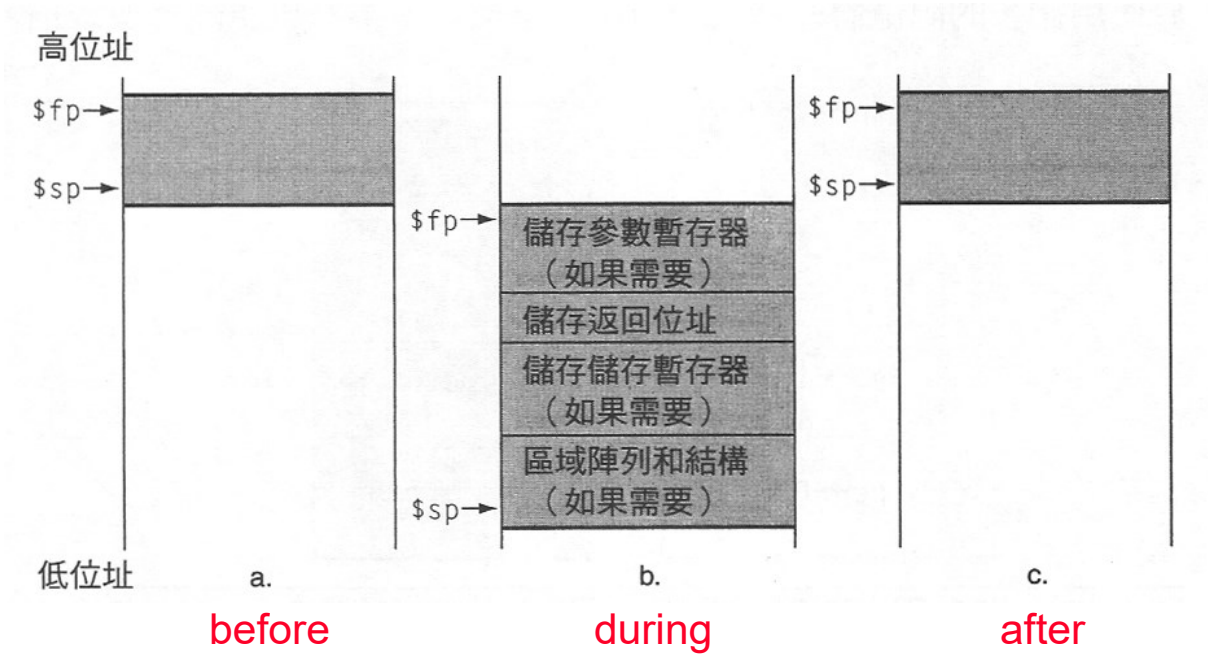
Name	Register number	Usage
`$zero`	0	the constant value 0
`$v0-$v1`	2-3	values for results and expression evaluation
`$a0-$a3`	4-7	arguments
`$t0-$t7`	8-15	temporaries
`$s0-$s7`	16-23	saved
`$t8-$t9`	24-25	more temporaries
`$gp`	28	global pointer
`$sp`	29	stack pointer
`$fp`	30	frame pointer
`$ra`	31	return address

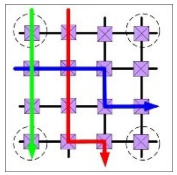
Register 1 ($at) reserved for assembler, 26-27 for operating system



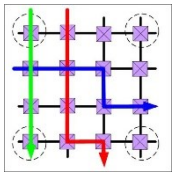
Supporting Procedures

- *What happens if the numbers of arguments and return results are more than 4 and 2, respectively?*
  - ***Stack***





Outline



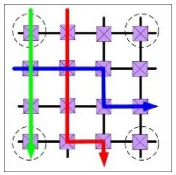
Constants

- *Small constants* are used quite frequently (50% of operands)

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

- *Solutions*?
  - put 'typical constants' **in memory** and load them (p. 85)
  - create **hard-wired registers** (like $zero) for constants like one

- ***Why not above solutions?***
  - ***Immediate instructions are better (addi, subi, …)***
    - ***But, ….. limited by 16-bit width***



Constants

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

- `add $29, $29, $28`

	Rs ($29, 32 bits)
+	Rt ($28, 32 bits)
	Rd ($29, 32 bits)

How about larger constants?

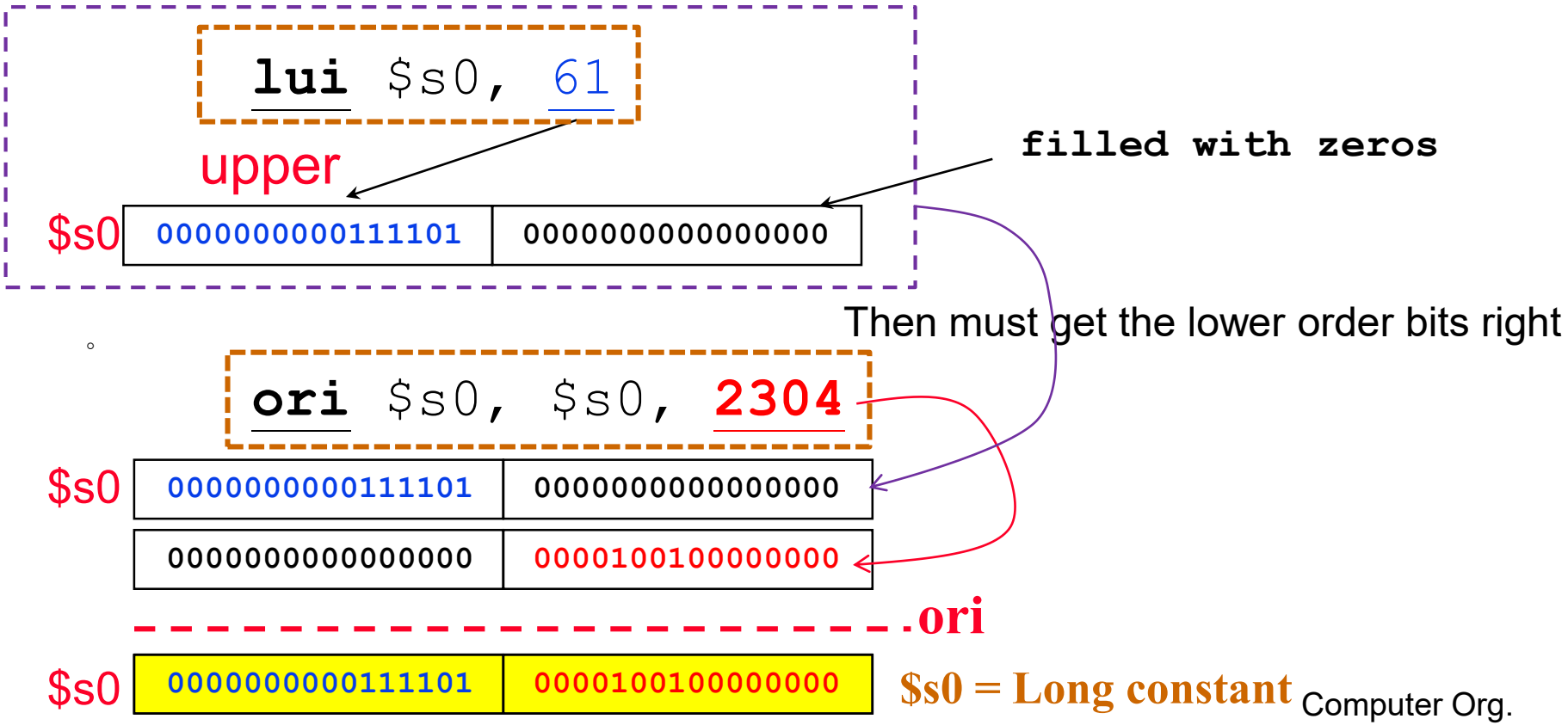
- `addi $29, $29, 10`

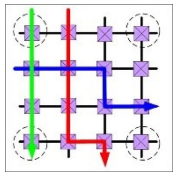
	rs ($29, 32 bits)
+	10 (? bits)
	rt ($29, 32 bits)

At most 16-bit constant

How about larger constants? (p.107 example)

- We'd like to be able to manually load a **32-bit constant** into a register (**Ex. 0000 0000 0011 1101 0000 1001 0000 0000**)
- Must use two instructions, new "*load upper immediate*" instruction



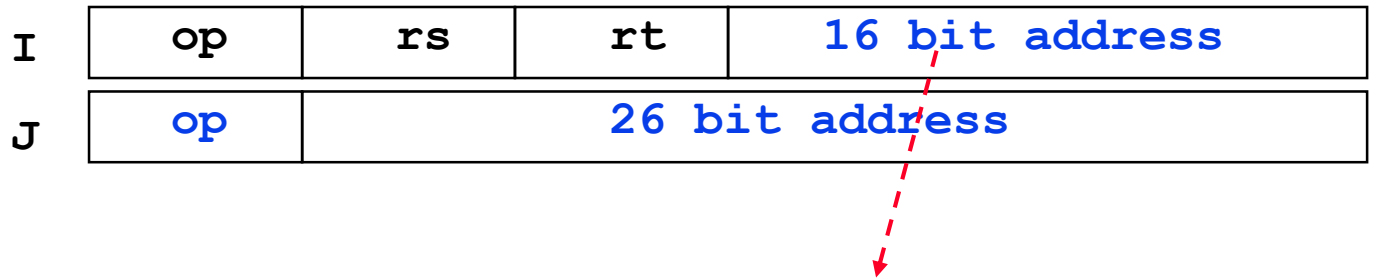


Addresses in Branches and Jumps

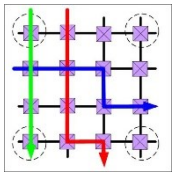
- Instructions:

`bne $t4,$t5,Label`	Next instruction is at Label if `$t4 != $t5`
`beq $t4,$t5,Label`	Next instruction is at Label if `$t4 = $t5`
`j Label`	**Next instruction is at Label**

- Formats:

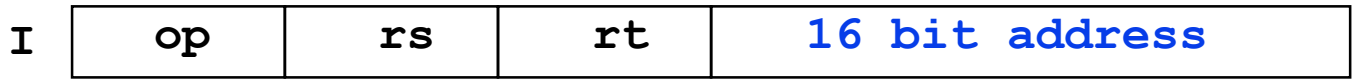


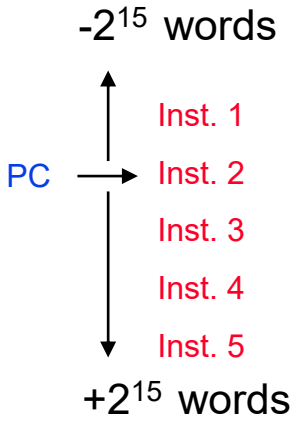
- No program could be bigger than 2^{16}
  — ***How do we handle addresses bigger than 2^{16}?***
  - ***Program counter = Register + Branch address***



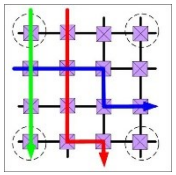
Addresses in Branches

- Could *specify a register* (like lw and sw) and add it to address
  - ***PC-relative addressing***
  - use Instruction Address Register (**PC** = program counter)
  - *most branches are local* (principle of locality)
- *MIPS stretches the distance of the branch by having PC-relative addressing refer to the number of words to the next instruction*



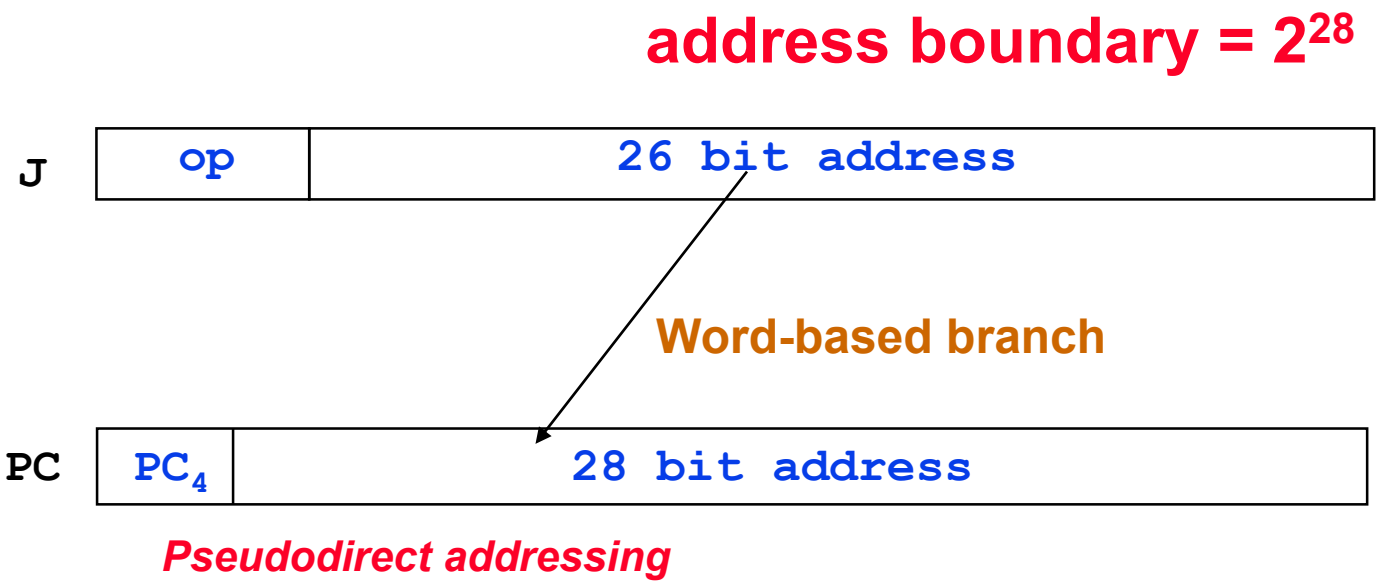


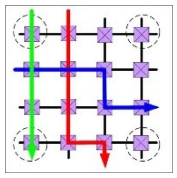
address boundary = 2^{16} words



Addresses in Jump

- Branching far away
- Jump instructions just *use high order bits of PC*
  - address boundaries of 256 MB (2^{28})



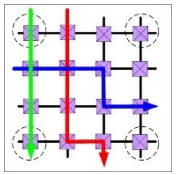


Example – p. 110

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

addr.	Machine code					
80000	0	0	19	9	2	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	**2**		
80016	8	19	19	1		
80020	2	**20000**				
80024						

- *bne branch address = 2 x 4 + PC(80016) = 80024*
- *jump address = 20000 x 4 = 80000*



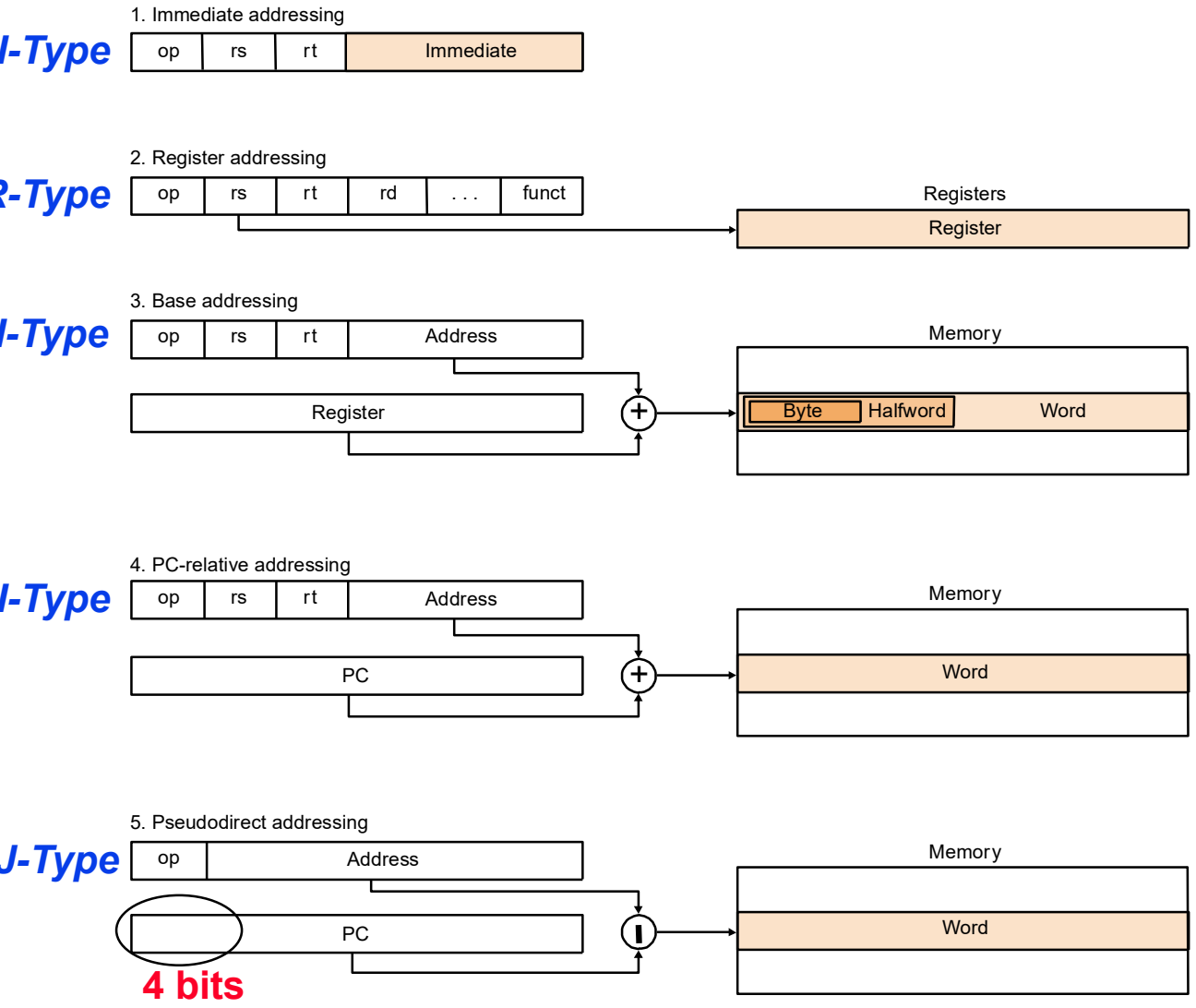
To summarize: Instructions

MIPS operands		
Name	**Example**	**Comments**
	`$s0-$s7, $t0-$t9, $zero,`	Fast locations for data. In MIPS, data must be in registers to perform
32 registers	`$a0-$a3, $v0-$v1, $gp,`	arithmetic. MIPS register $zero always equals 0. Register $at is
	`$fp, $sp, $ra, $at`	reserved for the assembler to handle large constants.
	Memory[0],	Accessed only by data transfer instructions. MIPS uses byte addresses, so
2^{30} memory	Memory[4], ...,	sequential words differ by 4. Memory holds data structures, such as arrays,
words	Memory[4294967292]	and spilled registers, such as those saved on procedure calls.

MIPS assembly language

Category	Instruction	Example	Meaning	Comments
Arithmetic	add	`add $s1, $s2, $s3`	`$s1 = $s2 + $s3`	Three operands; data in registers
	subtract	`sub $s1, $s2, $s3`	`$s1 = $s2 - $s3`	Three operands; data in registers
	add immediate	`addi $s1, $s2, 100`	`$s1 = $s2 + 100`	Used to add constants
Data transfer	load word	`lw $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Word from memory to register
	store word	`sw $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Word from register to memory
	load byte	`lb $s1, 100($s2)`	`$s1` = Memory[`$s2` + 100]	Byte from memory to register
	store byte	`sb $s1, 100($s2)`	Memory[`$s2` + 100] = $s1	Byte from register to memory
	load upper immediate	`lui $s1, 100`	$s1 = 100 * 2^{16}	Loads constant in upper 16 bits
Conditional branch	branch on equal	`beq $s1, $s2, 25`	if (`$s1 == $s2`) go to PC + 4 + 100	Equal test; PC-relative branch
	branch on not equal	`bne $s1, $s2, 25`	if (`$s1 != $s2`) go to PC + 4 + 100	Not equal test; PC-relative
	set on less than	`slt $s1, $s2, $s3`	if (`$s2 < $s3`) `$s1` = 1; else `$s1` = 0	Compare less than; for beq, bne
	set less than immediate	`slti $s1, $s2, 100`	if (`$s2 < 100`) `$s1` = 1; else `$s1` = 0	Compare less than constant
Uncondi-tional jump	jump	`j 2500`	go to 10000	Jump to target address
	jump register	`jr $ra`	go to `$ra`	For switch, procedure return
	jump and link	`jal 2500`	`$ra` = PC + 4; go to 10000	For procedure call

To summarize: Addressing modes



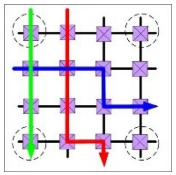
Immediate
Arithmetic inst.
Ex. addi

Arithmetic inst.
Ex. add

Load/Store inst.

Branch inst.

Jump inst.

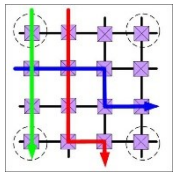


Overview of MIPS

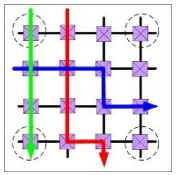
- simple instructions all 32 bits wide
- very structured, no unnecessary baggage
- only three instruction formats

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address/const.		
J	op	26 bit address				

- *rely on compiler* to achieve performance
  — what are the compiler's goals?
- help compiler where we can



Outline

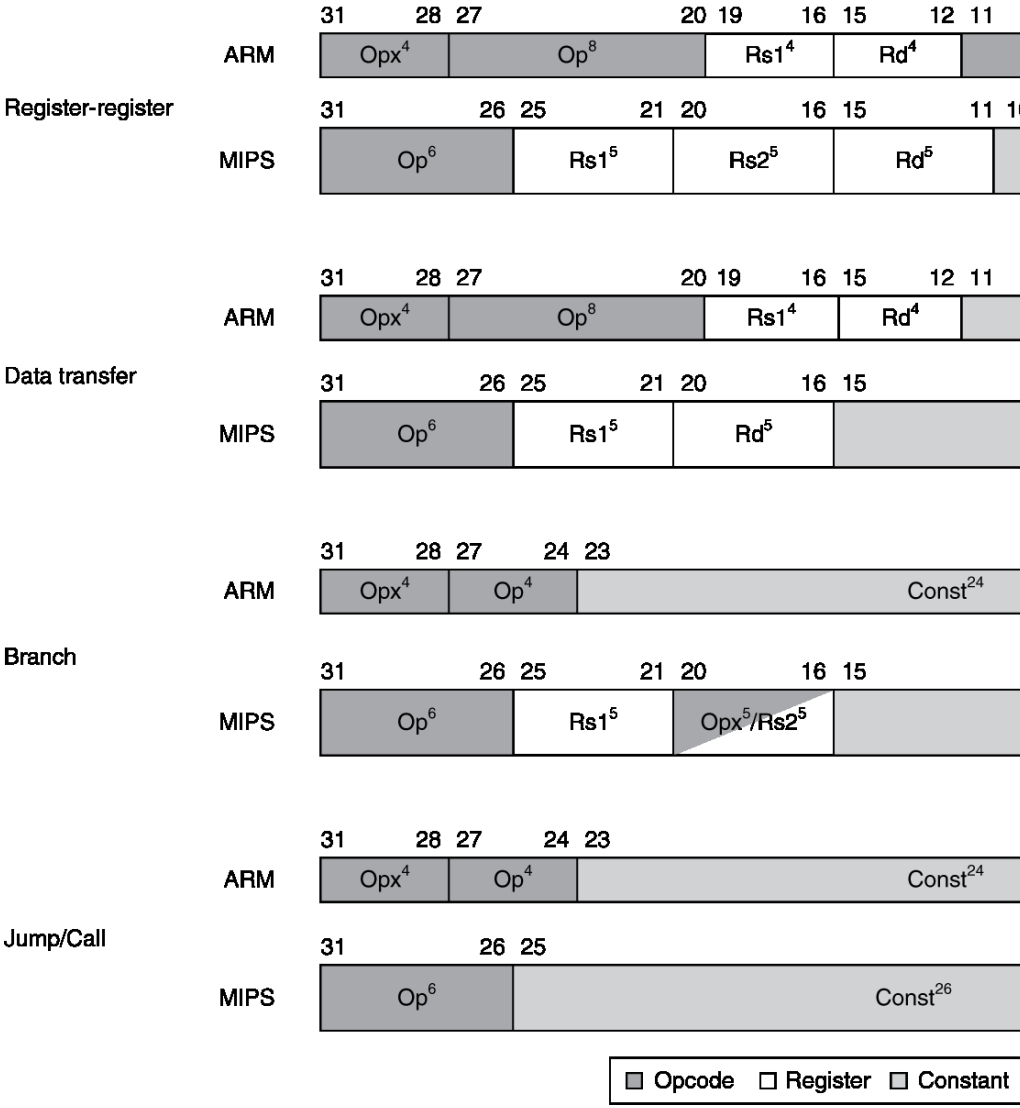


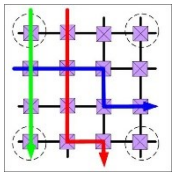
ARM & MIPS Similarities

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

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

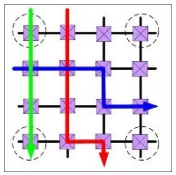
Instruction formats ：ARM vs. MIPS





X86 Architectures

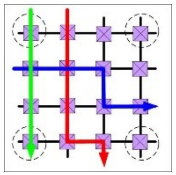
- Design alternative:
  - *provide more powerful operations*
  - goal is to reduce number of instructions executed
  - danger is a slower cycle time and/or a higher CPI
    - ***"The path toward operation complexity is thus fraught with peril. To avoid these problems, designers have moved toward simpler instructions"***



X86 history

- 1978: The *Intel 8086 is announced* (16 bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486, Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: *57 new "MMX" instructions are added, Pentium II*
- 1999: The Pentium III added another 70 instructions (SSE)
- 2001: Another 144 instructions (SSE2)
- 2003: *AMD extends the architecture to increase address space to 64 bits, widens all registers to 64 bits and other changes (AMD64)*
- 2004: Intel capitulates and embraces AMD64 (calls it EM64T) and adds more media extensions

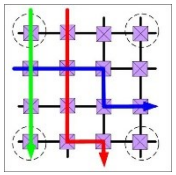
- "This history illustrates the impact of the "golden handcuffs" of compatibility
- "Adding new features as someone might add clothing to a packed bag"
- The is difficult to explain and impossible to love"



X86 Overview

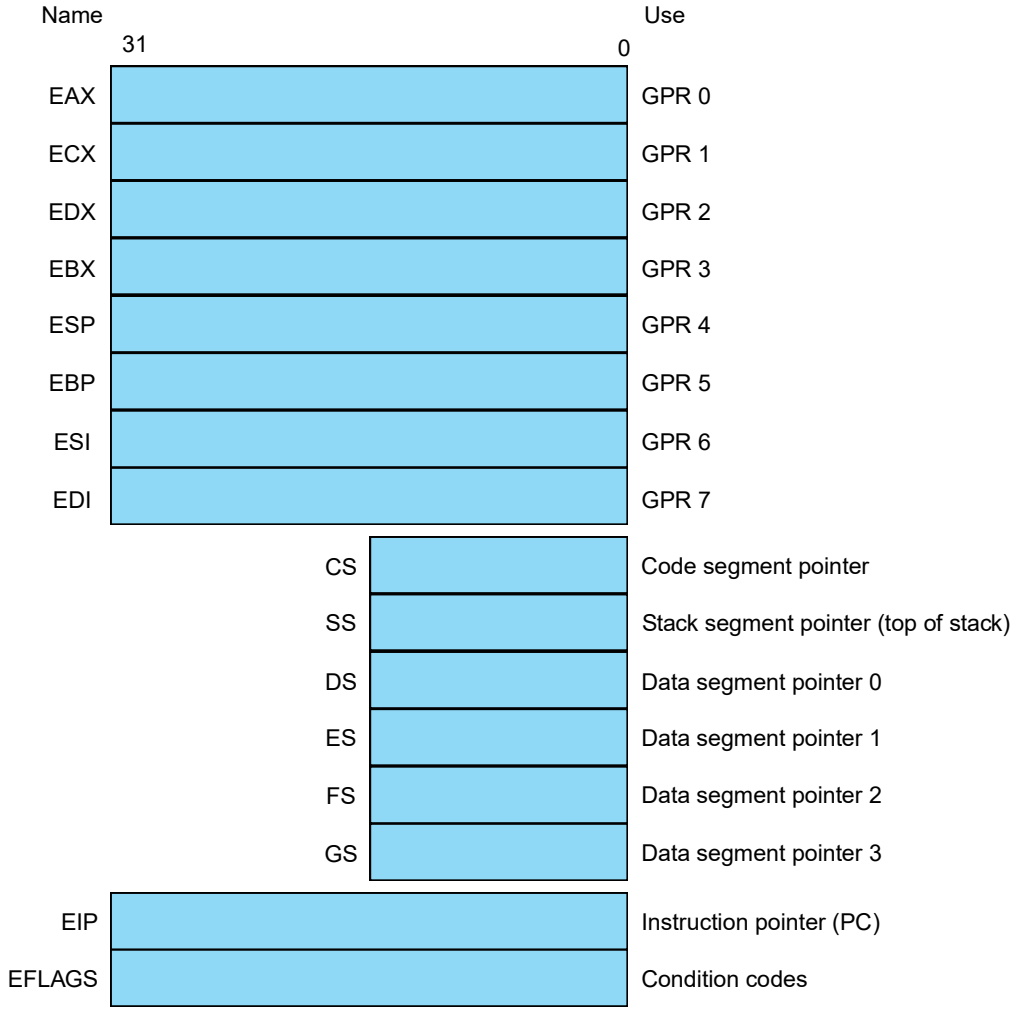
- Complexity:
  - *Instructions from 1 to 17 bytes long*
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

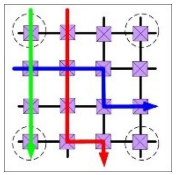
“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”



X86 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386





X86 Register Restrictions

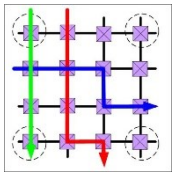
- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register Indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)#≤16-bit #displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)#≤16-bit #displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

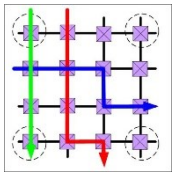


X86 Typical Instructions

- Four major types of integer instructions:
  - Data movement including **move, push, pop**
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

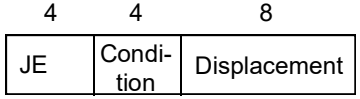
FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)



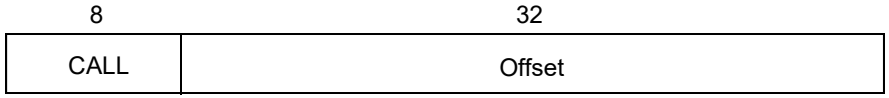
X86 instruction Formats

- Typical formats: (***notice the different lengths***)

a. JE EIP + displacement

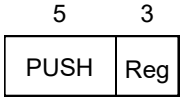


b. CALL

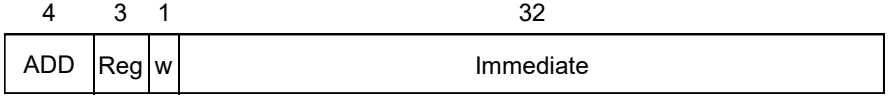


c. MOV EBX, [EDI + 45]

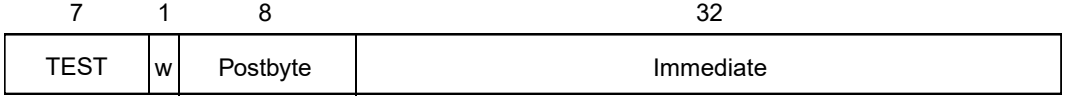
d. PUSH ESI

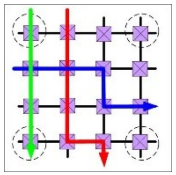


e. ADD EAX, #6765



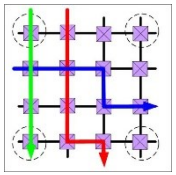
f. TEST EDX, #42





Summary - I

- Instruction complexity is only one variable
  - *lower instruction count vs. higher CPI / lower clock rate*
- Design Principles:
  1. simplicity favors regularity
  2. smaller is faster
  3. good design demands compromise
  4. make the common case fast
- Instruction set architecture (ISA)
  - a very important abstraction indeed!



Summary - II

RISC (Reduced Instruction Set Computer)	CISC (Complex Instruction Set Computer)
High-level: Keep operations simple to allow fast clock	High-level: Implement complex instructions to reduce # instructions in programs
1. Load-store architecture	1. Arithmetic instructions access memory
2. Fixed-width instruction encoding	2. Variable-width instruction encoding
3. Large register files	3. Have smaller register files
4. Long program code	4. Short program code
5. Complex compiler	5. Complex HW/Micro program

Q & A?