Computer Organization

Instruction Set Architecture

July 2015

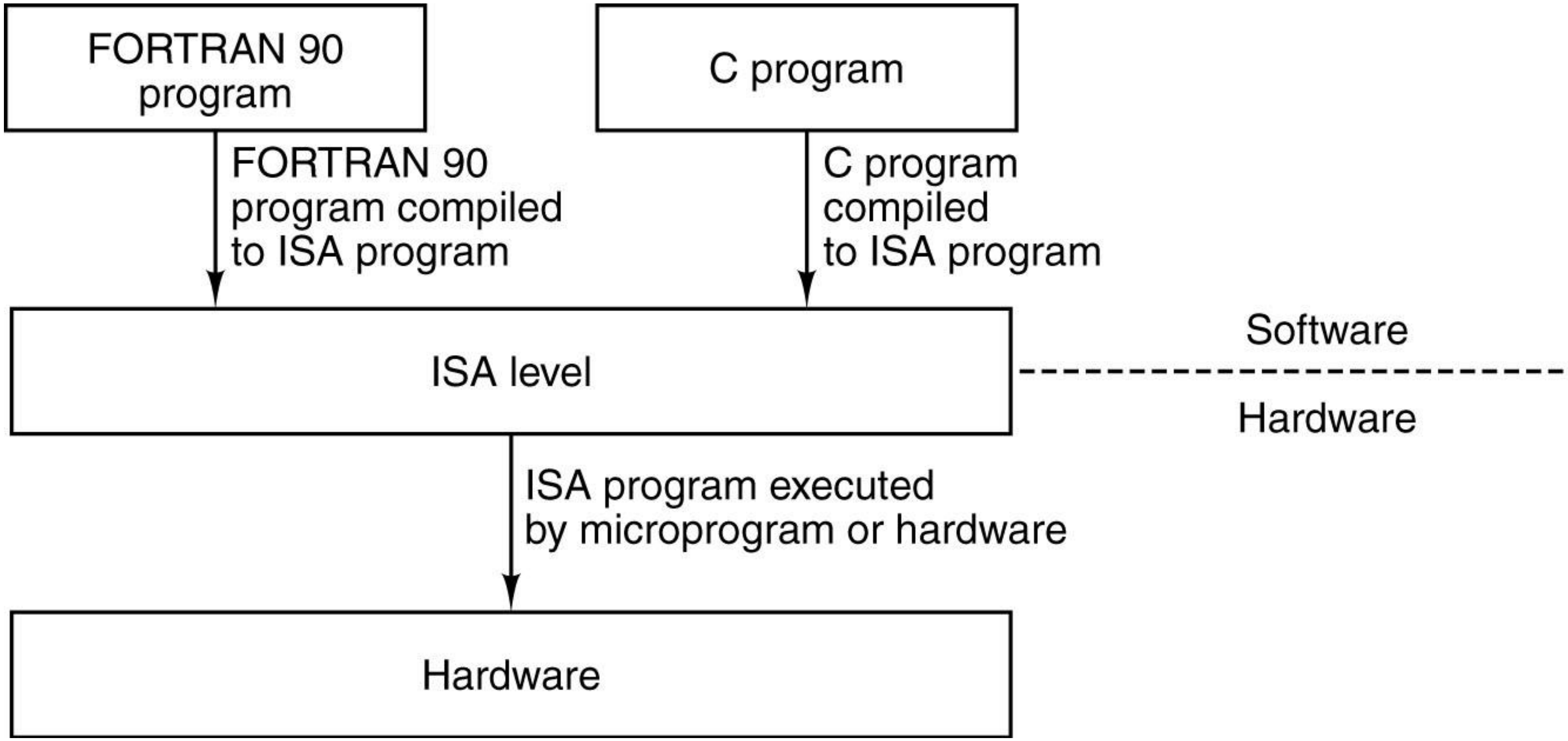
Why?

- Early days, new computer having its own with new set of instructions
- Needed to allow backward compatibility

Topics

- Instruction Set Architecture
- Key of ISA using MIPS
  - Design Principles
  - Instructions
  - Instruction formats
  - Addressing modes

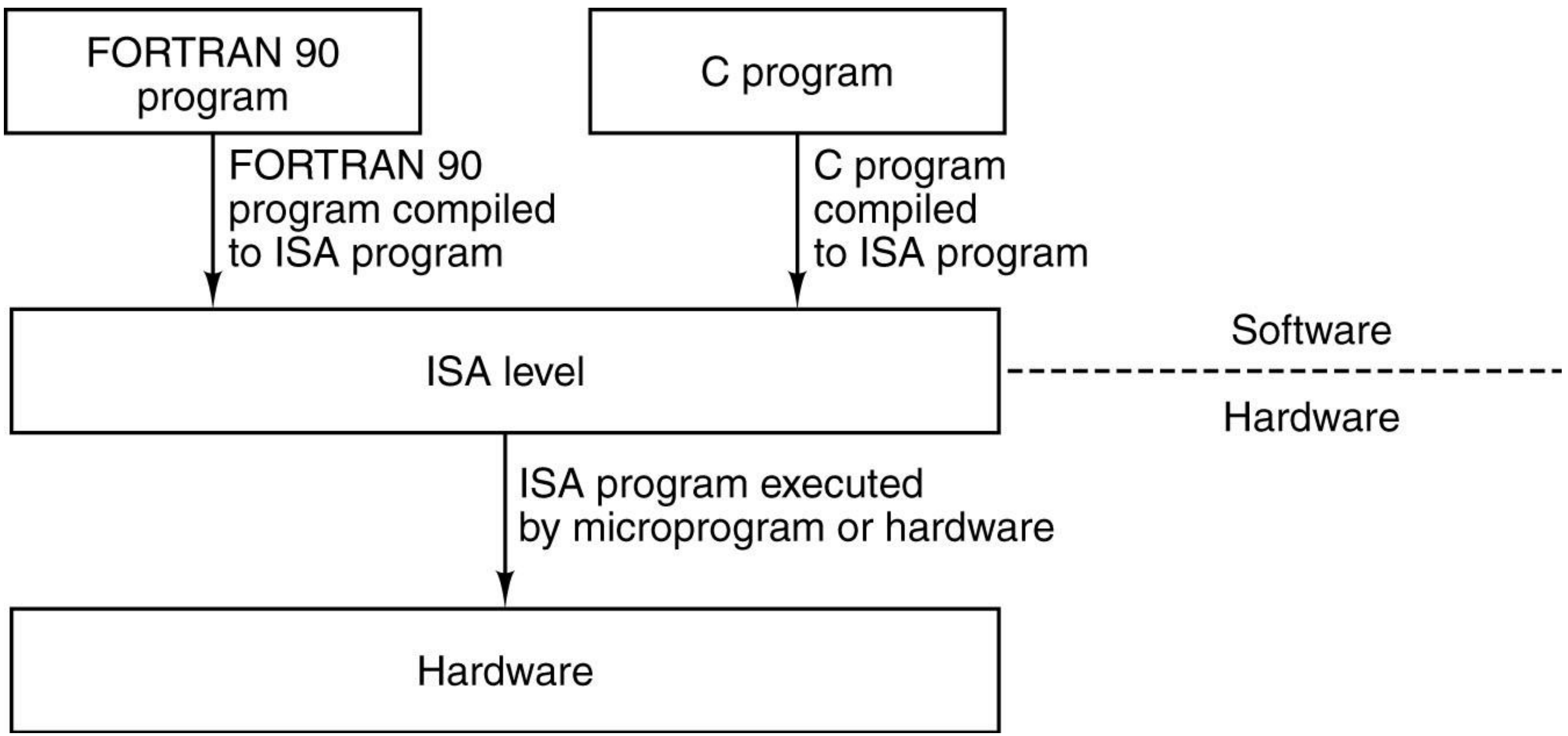
ISA


FORTRAN 90 program
C program
FORTRAN 90 program compiled to ISA program
C program compiled to ISA program
ISA level
Software
Hardware
ISA program executed by microprogram or hardware
Hardware

ISA or Instruction Set

- The level - between the high-level languages and the hardware
- When new hardware architecture comes along …
  - Can add new features to exploit new hardware capabilities
  - Need to maintain backward compatibility

ISA



ISA-level code is what a compiler outputs

ISA

- ISA-level code is what a compiler outputs
- Compiler writer needs to know:
  - Memory model
  - Types of registers are available
  - What instructions are available
    - instruction formats
    - opcodes
  - Exceptional conditions

ISA

- An ISA includes a specification of the set of opcodes (machine language), the native commands implemented by a particular processor

ISA

- It is distinguished from the microarchitecture
  - MAL which is the set of processor design techniques used to implement the instruction set
- Computers with different microarchitectures can share a common instruction set
- For example:
  - The Intel Pentium and the AMD Athlon implement nearly identical versions of the x86 instruction set, but have radically different internal designs

ISA

- Stored Program Concept
  - Fetch & Execute Cycle
    - Instructions are ***fetched*** and put into a special register
    - Bits in the register ***control*** the *subsequent actions* (= *execution*)
    - Fetch the next instruction and ***repeat***
- Instructions
  - Are encoded in binary, called machine code

Opcode	Operand Reference	Operand Reference

ISA Instructions

- More primitive than higher level languages,
  - e.g., no sophisticated control flow such as *while* or *for* loops
- Different computers have different instruction sets
  - But with many aspects in common
- Computers have very simple instruction sets
  - Makes the Implementation Simple

Instruction Set

- The complete collection of instructions that are understood by a CPU
  - Can be considered as a functional spec for a CPU
    - Implementing the CPU in large part is implementing the machine instruction set
- Machine Code is rarely used by humans
  - Binary numbers / bits
  - Usually represented by human readable assembly codes
  - In general, one assembler instruction equals one machine instruction

Elements of an Instruction

- Operation code (Op code)
  - Do this
- Source Operand reference
  - To this
- Result Operand reference
  - Put the result here
- Next Instruction Reference
  - When you have done that, do this...
  - Next instruction reference often implicit (sequential execution)

Operands

- Main memory (or virtual memory or cache)
  - Requires address
- CPU register
- I/O device
  - Several forms:
    - Specify I/O module and device
    - Specify address in I/O space
    - Memory-mapped I/O just another memory address

Sample Instruction Format

N bits

Opcode	Operand Reference	Operand Reference
X bits	Y bits	Z bits

Key of ISA

Operations

- What operations are provided??

Operands

- How many? how big?
- How are memory addresses computed?

How many registers?

Where do operands reside?

- e.g., can you add contents of memory to a register?

Instruction length

- Are all instructions of the same length?

Instruction format

- Which bits designate for what purpose??

Operations OR Instruction Types

- Data processing
  - Arithmetic and logical instructions
- Data storage (main memory)
- Data movement (I/O)
- Program flow control
  - Conditional and unconditional branches
  - Call and Return

ISA Architecture Types

Classification according to,

- Type of INTERNAL STORAGE in CPU
- Type and no. of OPERANDS

ISA Architecture Types

- In the CPU, type of INTERNAL STORAGE is the most basic differentiation in ISA
  - Stack, Accumulator or Set of registers
- Accordingly architectures are named:
  - Stack architecture
  - Accumulator architecture
  - Register architecture

ISA Architecture Types

- Operands may be named explicitly or implicitly
  - Stack architecture
    - Implicitly on the top of the stack
  - Accumulator architecture
    - One operand is implicitly the accumulator
  - General-purpose register architectures
    - Only explicit operands—either registers or memory locations
    - Operands may be accessed directly from memory or may need to be first loaded into temporary storage, depending on the class of instruction and choice of specific instruction

ISA

- Classification of Architectures according to the type and no. of operands
  - Load-store or register-register machines
    - With no memory reference per ALU instruction
  - Register-memory
    - Instructions with one memory operands per typical ALU instruction
  - Memory-memory
    - Instructions with one or more than one memory operand

ISA ISA Architecture Types

- Code C=A+B,
- On these three classes of instruction sets where A, B and C all belong in Memory

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store

ISA ISA Architecture Types

- Code C=A+B,
- On these three classes of instruction sets where A, B and C all belong in Memory

1. Stack	**2. Accumulator**	**3. Register**	
		Register-Memory	**Load-Store**
Push A Push B Add Pop C			

ISA ISA Architecture Types

- Code C=A+B,
- On these three classes of instruction sets where A, B and C all belong in Memory

1. Stack	**2. Accumulator**	**3. Register**	
		Register-Memory	**Load-Store**
Push A Push B Add Pop C	Load A Add B Store C		

ISA

Classes of register architecture

3.1 Register-memory architecture

Can access memory as part of any instruction

3.2 Load-store or register-register architecture

3.3 Memory-memory architecture

1. Stack	2. Accumulator	3. Register	
		Register-Memory	**Load-Store**
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	

ISA

Classes of register architecture

3.1 Register-memory architecture

Can access memory as part of any instruction

3.2 Load-store or register-register architecture

Can access memory only with load and store instructions

3.3 Memory-memory architecture

<table>
<tr><th rowspan="2">1. Stack</th><th rowspan="2">2. Accumulator</th><th colspan="2">3. Register</th></tr>
<tr><th>Register-Memory</th><th>Load-Store</th></tr>
<tr><td>Push A
Push B
Add
Pop C</td><td>Load A
Add B
Store C</td><td>Load R1, A
Add R1, B
Store C, R1</td><td>Load R1, A
Load R2, B
Add R3, R1, R2
Store C, R3</td></tr>
</table>

ISA

- Third class of register architecture
  3.3 Memory-Memory architecture
  - Keeps all operands in memory
  - **Not found in today's machines**

1. Stack	2. Accumulator	3. Register	
		Register-Memory	Load-Store
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

ISA

General Two classes of Register Architecture

3.1 Register-memory architecture

- Can access memory as part of any instruction

3.2 Load-store or register-register architecture

- Can access memory only with load and store instructions

1. Stack	2. Accumulator	3. Register	
		Register-Memory	**Load-Store**
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R1, B Store C, R1	Load R1, A Load R2, B Add R3, R1, R2 Store C, R3

Utilized in today's machine

ISA

- Example Code (A*B)–(C*D)–(E*F)
- On a stack architecture
  - Must be evaluated left to right, unless special operations or swaps of stack positions are done
  - A stack cannot be accessed randomly
- On an accumulator architecture
  - Creating lots of bus traffic
- On a register architecture
  - May be evaluated by multiplying in any order, which may be **more efficient** because of the location of the operands or because of pipelining

ISA

- Most Early Machines used
  - Stack or Accumulator-style architectures
  - Dedicating components / registers for special uses
    - Less number of general-purpose registers
    - Trying to allocate variables to registers will not be profitable

ISA-Load-Store Reg. Architecture

- Machines designed after 1980 uses a load-store register arch., the registers are used for variables
  - To reduce memory traffic
  - To speed up the program
    - As registers are **faster** than memory
  - To improve the code density
    - **Fewer bits** are needed to represent the register than the memory location
- Registers are easier for a compiler to use and can be used more effectively than other forms of internal storage

ISA-Load-Store Reg. Architecture

- How many registers are sufficient?
  - Answer depends on how they are used by the compiler
  - Most compilers reserve some registers for expression evaluation, use some for parameter passing, and allow the remainder to be allocated to hold variables

ISA

- GPR’s major concern-the type of operands for a typical arithmetic or logical instruction
  1. Whether ALU instruction has two or three operands
     - 3-operand instruction format
       - Instruction contains a result and two source operands
     - 2-operand instruction format
       - One of the operands is both a source and a result for the operation
  2. How many of the operands may be memory addresses in ALU instructions
     - May vary from none to three

ISA

- Classification of Architectures according to the type and no. of operands
  - Load-store or register-register machines
    - With no memory reference per ALU instruction
  - Register-memory
    - Instructions with one memory operands per typical ALU instruction
  - Memory-memory
    - Instructions with with one or more than one memory operand

ISA – GPR Architecture

1) Register-register (0-Memory + 3-Reg = Total 3)

- Advantage
  - Simple, fixed-length instruction encoding
  - Simple code-generation model
  - Instructions take similar numbers of clocks to execute
- Disadvantage
  - Higher instruction count than architectures having memory references in instructions
  - Some instructions are short and bit encoding may be wasteful
- Example - SPARC, MIPS, PowerPC, ALPHA

ISA – GPR Architecture

2) Register – memory (1- Memory + 1-Reg= Total 2)

- Advantage
  - Data can be accessed without loading first
  - Instruction format tends to be easy to encode and yields good density
- Disadvantage
  - Operands are not equivalent since a source operand in a binary operation is destroyed
  - Encoding a register number and a memory address in each instruction may restrict the number of registers
  - Clocks per instruction varies by operand location
- Example - Intel 80x86, Motorola 68000

ISA – GPR Architecture

3) Memory-memory (3-Memory + 0-Reg = Total-3)

- Advantage
  - Most compact
  - Doesn’t waste registers for temporaries
- Disadvantage
  - Large variation in instruction size, especially for three-operand instructions
  - Also, large variation in work per instruction
  - Memory accesses create memory bottleneck
- Example - VAX

ISA

- Summary, In general,
  - Machines with fewer alternatives make the **compiler's task simpler** since there are fewer decisions for the compiler to make
  - Machines with a wide variety of flexible instruction formats **reduce the number of bits** required to encode the program
  - A machine that uses a small number of bits to encode the program is said to have good instruction density—a smaller number of bits do as much work as a larger number on a different architecture
  - The no. of registers also affects the instruction size

Operands

- How many operands are supported?
  - 3 operands
  - 2 operands
  - 1 operand
  - 0 operand

No. of Operands

- 3 operands
  - Operand 1, Operand 2, Result
  - a = b + c;
  - add ax, bx, cx
  - May be a fourth address - next instruction (usually implicit)[not common]
- Instructions are long because 3 or more operands have to be specified

No. of Operands

- 2 Operands
  - One address doubles as operand and result
  - a = a + b
  - add ax, bx
  - Reduces length of instruction over 3-address format
  - Requires some extra work by processor
  - Temporary storage to hold some results

No. of Operands

- 1 Operand
  - Implicit second address
  - Usually a register (accumulator)
  - Common on early machines
- Used in some Intel x86 instructions with implied operands
  - mul ax
  - idiv ebx

No. of Operands

- 0 (zero) Operand
  - All addresses implicit
  - Uses a stack- X87 example c = a + b:
    - fld a ;push a
    - fld b ;push b
    - fadd ;st(1)   //a+b, pop stack
    - fstp c ;store and pop c
- Can reduce to 3 instructions:
  - fld a ;push a
  - fld b ;push b
  - faddp c ;add and pop

Computation of Y = (a-b) / (c + (d * e))

- Three Operands instructions
- Two Operandsinstructions
- One Operand instructions

Computation of Y = (a-b) / (c + (d * e))

- Three Operands instructions
  - **sub y,a,b**
  - **mul t,d,e**
  - **add t,t,c**
  - **div y,y,t**
- Two Operands instructions
  - **mov y,a**
  - **sub y,b**
  - **mov t,d**
  - **mul t,e**
  - **add t,c**
  - **div y,t**

Computation of Y = (a-b) / (c + (d * e))

- One Operand instructions
  - **load d**
  - **mul e**
  - **add c**
  - **store y**
  - **load a**
  - **sub b**
  - **div y**
  - **store y**

How Many Operands?

- More Operands
  - More complex instructions
  - More registers
    - Inter-register operations are quicker
  - Fewer instructions per program
  - More complexity in processor
- Fewer Operands
  - Less complex instructions
  - One address format however limits you to one register
  - More instructions per program
  - Less complexity in processor
    - Faster fetch/execution of instructions

Memory Organization

- Viewed as a large single-dimension array with access by *address*
- A memory address is an *index* into the memory array
- Two views of Memory
  - Byte Addressing
    - The index points to a byte of memory, and that the unit of memory accessed by a load/store is a byte
  - Word Addressing

0	**8 bits of data**
1	**8 bits of data**
2	**8 bits of data**
3	**8 bits of data**
4	**8 bits of data**
5	**8 bits of data**
6	**8 bits of data**

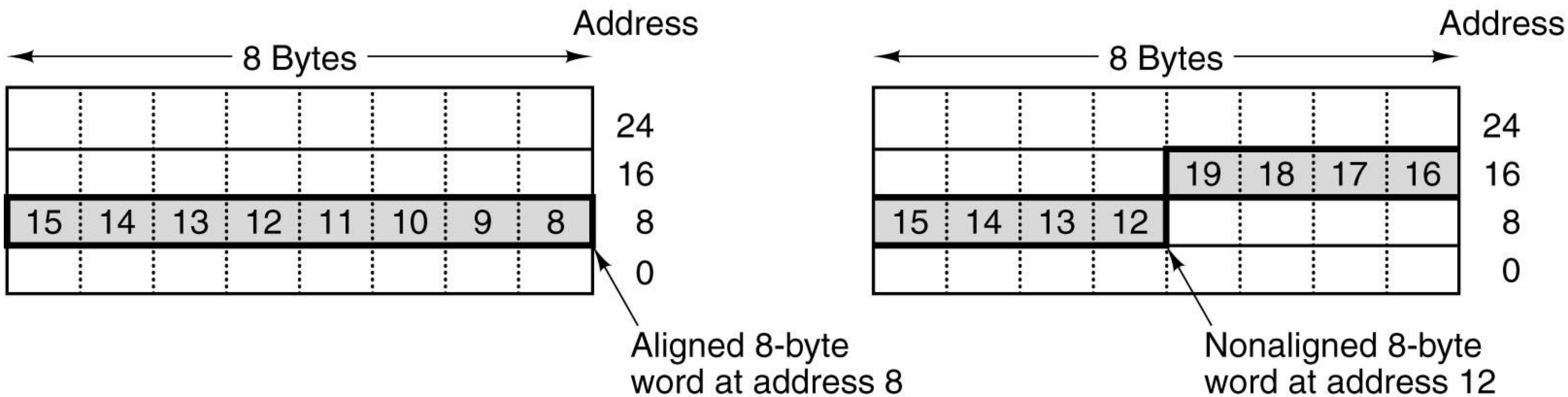
...

Memory Organization

- How many bytes (8 bits) and words (32 bits) can be accessed for 4 GB Memory?
  - 2^{32} bytes with byte addresses from 0 to 2^{32}-1
  - 2^{30} words with byte addresses 0, 4, 8, ... 2^{32}-4
    - <u>Words are *aligned*</u>

Memory Organization

- Why Word alignment ?
  - Memories operate more efficiently this way
- Consider 8-byte (64-bit) words



(a) (b)

Memory Organization

- Bytes in a word can be numbered in two ways:
  - Big Endian
    - Most-significant byte at least address of a word
      - MIPS is Big Endian
  - Little Endian
    - Least-significant byte at least address
      - ? Is Little Endian

Memory Organization

- Example: Store the number 12 in 32 bits

There will be 28 zeroes and then 1100

(MSB) 00000000 00000000 00000000 00001100 (LSB)

	Big-endian	Little-endian
Byte 0:	0000 0000	0000 1100
1:	0000 0000	0000 0000
2:	0000 0000	0000 0000
3:	0000 1100	0000 0000

The big-endian system 1100 is in **byte 3**
The little-endian system 1100 is in **byte 0**

ISA

- Example ISA's:
  - Digital's VAX (1977)
  - Intel's x86 (1978), but successful (IBM PC)
  - MIPS – focus of text, used in assorted machines
  - PowerPC – used in Mac's, IBM supercomputers, ...
- VAX and x86 are CISC ("Complex Instruction Set Computers")
  - Started in 70's
- MIPS and PowerPC are RISC ("Reduced Instruction Set Computers")
  - Almost all machines of 80's and 90's are RISC
    - Including VAX's successor, the DEC Alpha

RISC vs. CISC

RISC

- Simple instructions, few in number
- Instruction length - Fixed
- Instructions to access memory
 - only **LOAD/STORE**
- Addressing modes - Few
- Complexity in compiler
- Achieves shorten execution time by reducing the *clock cycles per instruction* (i.e. simple instructions take less time to interpret)

CISC

- Many complex instructions
- Instruction length - Variable
- Instructions to access memory - many instructions can access
- Addressing modes - Many
- Complexity in microcode
- Achieves shorten execution time by reducing the number of instructions per program

Example for RISC vs. CISC

Multiplication:

CISC:	Mov ax,10	RISC:	Mov ax,0
	Mov bx,5		Mov bx,10
	Mul bx, ax		Mov cx, 5
			Begin: **Add ax,bx**
			Loop begin

- The total clock cycles for the CISC version might be:
  (2 movs × 1 cycle) + (1 mul × 30 cycles)
  = 32 cycles

- While the clock cycles for RISC version is:
  (3 movs × 1 cycle) + (5 adds × 1 cycle) + (5 loops × 1 cycle)
  = 13 cycles

ISA

Design goals:

- *Maximize performance*
- *Minimize cost*
- *Reduce design time*

The MIPS

Microprocessor without **I**nterlocked **P**ipeline **S**tages

- RISC instruction set architecture (ISA)
- Large share of embedded core market
  - Applications in consumer electronics, network / storage equipment, cameras, printers, …
- Typical of many modern ISAs

MIPS Instruction Set

- What should be considered?
  - Operations (MIPS Arithmetic)
  - MIPS Operand
    - Register
    - Memory

Operations (MIPS Arithmetic)

Example:

- C code: A = B + C
- C code: A = B + C + D + E
- C code: F = (G + H) – (I + J)
- C code: G = H + A[8];

Operations (MIPS Arithmetic)

Example:

- C code: A = B + C

- All MIPS arithmetic instructions have 3 operands
- Operand order is fixed (e.g., destination first)

- MIPS code: Add A, B, C

MIPS Arithmetic

Example:

- C code: A = B + C + D + E

- MIPS code:

  Add A, B, C

  Add A, A, D

  Add A, A, E

MIPS Arithmetic

Example:

- C code: F = (G + H) – (I + J)

- MIPS code:

  Add F, G, H

  Sub F, I, J

MIPS Arithmetic

Example:

- C code: F = (G + H) – (I + J)

- MIPS code: //Use of temporary variables

```
Add $t0, G, H
Add $t1, I, J
Sub F, $t0, $t1
```

MIPS Arithmetic

Design Principle 1:

- Simplicity favors regularity.
  - i.e. Regularity makes implementation simpler

- Simplicity enables higher performance at lower cost

MIPS Operand

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

MIPS Operand

- Only 32 Registers?

Design Principle 2:

- Smaller is faster.
- *Why?*
  - Electronic signals have to travel further on a physically larger chip increasing clock cycle time
  - Smaller is also cheaper!

MIPS Register Operand

- C code:

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

- Compiled MIPS code:
  - f, …, j in \$s0, …, \$s4

  Add \$t0, \$s1, \$s2

  Add \$t1, \$s3, \$s4

  Sub \$s0, \$t0, \$t1

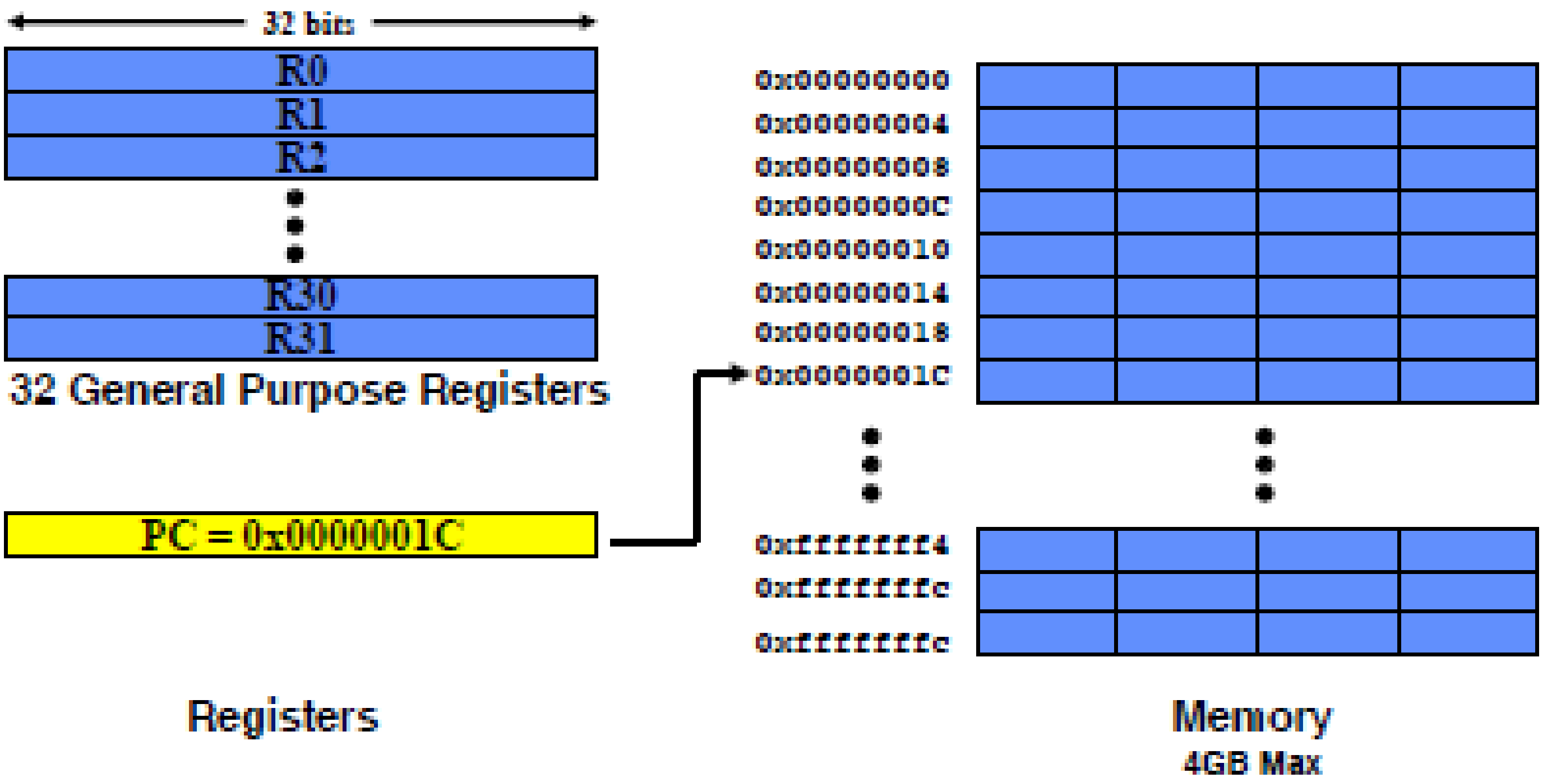
MIPS Register Operand

- Arithmetic instructions operands must be in registers
  - MIPS has only 32 registers
- Compiler associates variables with registers
- What about programs with lots of variables (arrays, etc.)?

MIPS Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory

MIPS Registers and Memory



MIPS Memory Organization

- Memory is byte addressed
  - Each address identifies an 8-bit byte
  - A word is 32 bits or 4 bytes
  - Address must be a multiple of 4
  - <u>Words are aligned</u> in memory
  - Follows Big-Endian Ordering

MIPS Load/Store Instructions

- Load word has destination first
- Store has destination last

- MIPS arithmetic operands are registers, not memory locations
  - Therefore, words must first be moved from memory to registers using loads before they can be operated on; then result can be stored back to memory

MIPS Load/Store Instructions

- C code:

  G = H + A[8];

  G in $s1, H in $s2, base address of A in $s3

- Compiled MIPS code:

  Index 8 requires offset of 32, due to 4 bytes/word

```
Value        Offset      Base address
Lw $t0, 32($s3)          # load word
Add $s1, $s2, $t0
```

MIPS Load/Store Instructions

- C code:

  A[12] = H + A[8];

- MIPS code:

  ?

Load	:	Lw $t0, 32($s3)
Arithmetic	:	Add $t0, $s2, $t0
Store	:	Sw $t0, 48($s3)

MIPS Load/Store Instructions

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

Instruction Format: R Type

MIPS Operand - Register

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, called \$at, is reserved for the assembler; registers 26-27, called \$k0 and \$k1 are reserved for the operating system

- Require 5 bits to select one register

Instruction Format: R Type

- Opcode and Operand
  - 3 Registers Operands
    - 15 bits for Register Operands
  - Opcode

Instruction Format: R Type

op	rs	rt	rd	shamt	funct
opcode – operation	first register source operand	second register source operand	register destin-ation operand	shift amount 00000 for now	function field - selects variant of operation extends opcode
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

R Type Format Example

op	rs	rt	rd	shamt	funct
opcode – operation	first register source operand	second register source operand	register destin-ation operand	shift amount 00000 for now	function field - selects variant of operation extends opcode
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

Add $t0, $s1, $s2

$t0 – $t7: Registers are: 8 – 15
$t8 – $t9: Registers are: 24 – 25
$s0 – $s7:Registers are: 16 – 23

special	$s1	$s2	$t0	0	add
0	17	18	8	0	32
000000	10001	10010	01000	00000	100000

$00000010\ 00110010\ 01000000\ 00100000_2 = 02\ 32\ 40\ 20_{16}$

R Type Format Example

Sub $t0, $s1, $s2

$t0 – $t7: Registers are: 8 – 15
$t8 – $t9: Registers are: 24 – 25
$s0 – $s7:Registers are: 16 – 23

special	$s1	$s2	$t0	0	**sub**

0	17	18	8	0	**34**

000000	**10001**	**10010**	**01000**	**00000**	**100010**

00000010 00110010 01000000 00100000$_2$ = 02 32 40 22$_{16}$

Instruction Format

- Load Instruction
  - Lw $s1, 100($s2)
    - Two Registers
    - A Constant
      - If consider, the third register to store this
      - Would be limited to 5 bits only i.e. upto 32
      - This may be larger than 32
      - So, 5-bit field is too small

?

MIPS Instructions

Design Principle 3:

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

- Introduce a new type of instruction format with Immediate operands
  - Immediate operand avoids a load instruction

Instruction Format: I Type

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits
opcode – operation	first register source operand	second register source operand	constant: -2^{15} to $+2^{15} - 1$ address: offset added to base address in rs

Immediate Operands

- For example:
  - Constant data specified in an instruction
  - Addi $s3, $s3, 4
- No subtract immediate instruction
  - Just use a negative constant
  - Addi $s2, $s1, -1

Instruction Format: I Type

op	rs	rt	constant or address
6 bits	5 bits	5 bits	16 bits
opcode – operation	first register source operand	second register source operand	constant: -2^{15} to $+2^{15}-1$ address: offset added to base address in rs

lw $t0, 1002($s2)

$t0 – $t7: Registers are: 8 – 15
$t8 – $t9: Registers are: 24 – 25
$s0 – $s7:Registers are: 16 – 23

100011	10010	01000	0000001111101010

Example : I Type Format

- C code:

  A[300] = H + A[300];

- MIPS code:
  - Lw $t0, 1200($t1)
  - Add $t0, $s2, $t0
  - Sw $t0, 1200($t1)

Example: I Type Format

<table>
<tr><th>op</th><th>rs</th><th>rt</th><th>rd</th><th>shamt/
address</th><th>funct</th></tr>
<tr><td>35</td><td>9</td><td>8</td><td colspan="3">1200</td></tr>
<tr><td>0</td><td>18</td><td>8</td><td>8</td><td>0</td><td>32</td></tr>
<tr><td>43</td><td>9</td><td>8</td><td colspan="3">1200</td></tr>
</table>

$t0 – $t7: Registers are: 8 – 15
$t8 – $t9: Registers are: 24 – 25
$s0 – $s7:Registers are: 16 – 23

Lw $t0, 1200($t1)
Add $t0, $s2, $t0
Sw $t0, 1200($t1)

Logical Operations

- Instructions for bitwise manipulation

<table>
<tr><th>Operation</th><th>C</th><th>MIPS</th></tr>
<tr><td>Shift Left</td><td><<</td><td>sll</td></tr>
<tr><td>Shift Right</td><td>>></td><td>srl</td></tr>
<tr><td>Bitwise AND</td><td>&</td><td>and, andi</td></tr>
<tr><td>Bitwise OR</td><td>|</td><td>or, ori</td></tr>
<tr><td>Bitwise NOT</td><td>~</td><td>nor</td></tr>
</table>

- Useful for extracting and inserting groups of bits in a word

Shift Operations

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

- 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)

AND Operations

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

And $t0, $t1, $t2 # $t0 = $t1 & $t2

$t2 = 0000 0000 0000 0000 00<u>00 11</u>01 1100 0000

$t1 = 0000 0000 0000 0000 00<u>11 11</u>00 0000 0000

$t0 = 0000 0000 0000 0000 00<u>00 11</u>00 0000 0000

OR Operations

- Useful to include bits in a word
  - Select some bits to 1, leave others unchanged
- Example:

Or $t0, $t1, $t2 # $t0 = $t1 | $t2

$t2 = 0000 0000 0000 0000 00<u>00 11</u>01 1100 0000

$t1 = 0000 0000 0000 0000 00<u>11 11</u>00 0000 0000

$t0 = 0000 0000 0000 0000 00<u>11 11</u>01 1100 0000

The Constant Zero

- MIPS register 0 ($zero) is the constant 0
  - Cannot be overwritten
- Useful for common operations
  - E.g., move between registers:
    - Value of $s1 to $t2
    - Add $t2, $s1, $zero

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)
- Example:

Nor $t0, $t1, $zero # Register 0: always read as zero

$t1 = 0000 0000 0000 0000 00<u>11 11</u>00 0000 0000

$t0 = 1111 1111 1111 1111 11<u>00 00</u>11 1111 1111

Conditional Operations

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

MIPS conditional instructions

- 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:

  Bne $s3, $s4, Else

  Add $s0, $s1, $s2

  J Exit

  Else: Sub $s0, $s1, $s2

  Exit: …

Assembler calculates addresses

Compiling Loop Statements

- C code: (with Variable Array Index)

Loop: g = g + A[i];

i = i + j;

if (i != h) goto Loop;

Variables g, h, i and j to the registers $s1, $s2, $s3 and $s4

Assume A is an array of 100 elements and its base address is in $s5

C code to MIPS code

- C Code:

  g = h + A[i];

  - Assume A is an array of 100 elements and its base address is in $s3
  - Variables g, h and i to the registers $s1, $s2 and $s4

C (g = h + A[i];) to MIPS

Array A of 100 elements and its base address is in $s3
Variables G, H and i to the registers $s1, $s2 and $s4

- Load A[i] into a temporary register

Due to Byte Addressing, Must multiply i by 4

i.e. i + i = 2i and then 2i + 2i = 4i

- MIPS Code:

```
add $t1, $s4, $s4    # $t1 = 2 * i
add $t1, $t1, $t1    # $t1 = 4 * i
```

C (g = h + A[i];) to MIPS

Array A of 100 elements and its base address is in $s3
Variables G, H and i to the registers $s1, $s2 and $s4

- To get the address of A[i],

Need to add $t1 and the base of A in $s3

i.e., add $t1, $t1, $s3

$t1=address of A[i] (4 * i + $s3)

C (g = h + A[i];) to MIPS

- Now use Load A[i] into a temporary register

i.e., lw $t0, 0($t1)

$t0 = A[i]

Final Instruction adds A[i] and h, and places the sum in g:

i.e., add $s1, $s2, $t0

g = h + A[i]

C (g = h + A[i];) to MIPS

- **MIPS Code:**

```
add $t1, $s4, $s4    # $t1 = 2 * i
add $t1, $t1, $t1     # $t1 = 4 * I
add $t1, $t1, $s3     #  $t1=address of A[i] (4 * i + $s3)
lw $t0, 0($t1)         #  $t0 = A[i]
add $s1, $s2, $t0     # g = h + A[i]
```

Compiling Loop Statements

- C code: (with Variable Array Index)

Loop: g = g + A[i]; i = i + j; if (i != h) goto Loop;
Variables g, h, i and j to the registers $s1, $s2, $s3 and $s4
Array A of 100 elements and its base address is in $s5

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3   # Temp reg $t1 = 2 * i
      add $t1, $t1, $t1   # Temp reg $t1 = 4 * i
      add $t1, $t1, $s5   # $t1 = address of A[i]
      lw $t0, 0($t1)
      add $s1, $s1, $t0   # g = g +A[i]
      add $s3, $s3, $s4   # i = i + j
      bne $s3, $s2, Loop
```

Compiling While Loop

- C code:

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

  - i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code: ?

Compiling Loop Statements

- C code:

while (save[i] == k) i = i + j;

  - i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3    # Temp reg $t1 = 2 * i
      add $t1, $t1, $t1    # Temp reg $t1 = 4 * i
      add $t1, $t1, $s6    # $t1 = address of save[i]
      lw $t0, 0($t1)       # Temp reg $t0 = save[i]
      bne $t0, $s5, Exit   # Go to Exit if save[i] != k
      add $s3, $s3, $s4    # i = i + j
      j Loop               # Go to loop
Exit:
```

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

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:

  ?

Tutorial Question

- C code:

```
switch (k){
case 0: f = i + j; break;
case 1: f = g + h; break;
case 2: f = g - h; break;
case 3: f = i - j; break;
}
```

- Six variables f through k correspond to six registers $s0 through $s5
- Compiled MIPS code: **?**

Branch Instruction Format

- Instructions:
  - beq rs, rt, L1
  - bne rs, rt, L1
  - Specify:
    - Opcode, two registers, target address

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

- 16 bit Address ?

Branch Addressing

- 16 bits is too small a reach in a 2^{32} address space

- Solution:
  - Principle of locality
    - Most branch targets are near branch
    - Forward or backward Direction
  - Use PC (= program counter), called *PC-relative* addressing based on Principle of Locality
  - PC-relative addressing
    - Target address = PC + offset × 4
    - PC already incremented by 4 by this time

Target Addressing Example

- C code:

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

//i in $s3, j in $s4, k in $s5,

base address of save in $s6

Assume Loop at location **80000**

Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
Exit:
```

Assume Loop at location **80000**

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	?		
80020	0	19	20	19	0	32
80024	2	?				
80028	…					
80012	35	9	8	0		

Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
Exit:
```

Assume Loop at location **80000**

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	80028		
80020	0	19	20	19	0	32
80024	2	80000				
80028	…					

80012	35	9	8	0

Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
Exit:
```

Assume Loop at location **80000**

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	8		
80020	0	19	20	19	0	32
80024	2	80000				
80028	…					
80012	35	9	8	0		

Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
Exit:
```

Assume Loop at location **80000**

8000						
8000						
8000						
80012	35	9	8			
80016	5	8	21	2		
80020	0	19	20	19	0	32
80024	2	~~80000~~				
80028	…					
80012	35	9	8	0		

As the PC-Relative addressing refers the number of words to the next instruction instead of the number of bytes.
So, 8 bytes is replaced by 2 words

Jump Addressing

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

op	address
6 bits	26 bits

- Pseudo-Direct jump addressing
  - 26 bit address is concatenated with the upper bits of the PC
  - Target address = $PC_{31\ldots28}$: (address × 4)

Jump Addressing

- MIPS jump `j` instruction replaces *lower* 28 bits of the PC with `A00` where `A` is the 26 bit address; it *never changes* upper 4 bits
  - *Example*:
    if `PC = 1011X` (where X = 28 bits), it is replaced with `1011A00`
  - Why Not upper 4 bits?
  - Address space size = 2^{32}
    - There are 16(=2^4) partitions of the, each partition of size 256 MB (=2^{28}), *such that*, in each partition the upper 4 bits of the address is same.
  - If a program crosses an address partition, then a `j` that reaches a different partition has to be replaced by `jr` with a full 32-bit address first loaded into the jump register
  - Therefore, OS should always try to load a program inside a single partition

Jump Addressing

- Example:

  J Label # Address of Label = 100

  - 26-bit Pseudodirect address is 100/4 = 25

op	26 bit number
6 bits	26 bits
000010	00000000000000000000011001

Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
 Exit:
```

Assume Loop at location **80000**

80000	0	19	19	9	0	32
80004	0	9	9	9	0	32
80008	0	9	21	9	0	32
80012	35	9	8	0		
80016	5	8	21	2		
80020	0	19	20	19	0	32
80024	2	80000				
80028	…					
80012	35	9	8	0		

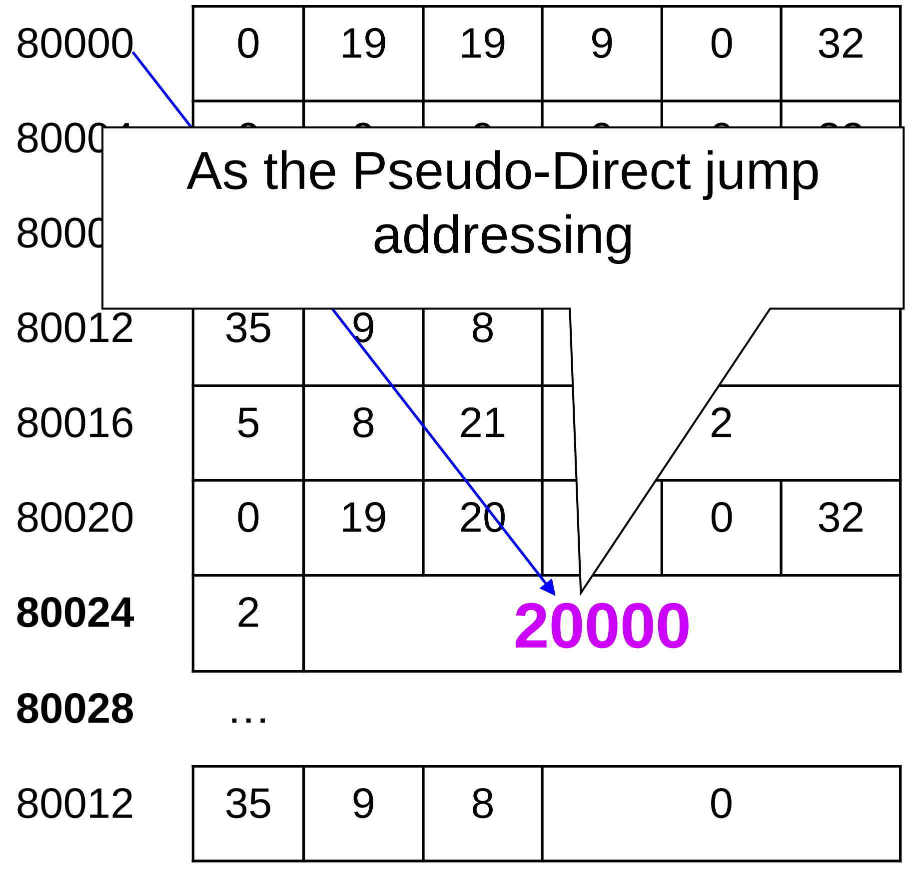
Target Addressing Example

C code: while (save[i] == k) i = i + j; i in $s3, j in $s4, k in $s5, base address of save in $s6

- Compiled MIPS code:

```
Loop: add $t1, $s3, $s3
      add $t1, $t1, $t1
      add $t1, $t1, $s6
      lw $t0, 0($t1)
      bne $t0, $s5, Exit
      add $s3, $s3, $s4
      j Loop
 Exit:
```

Assume Loop at location **80000**



More Conditional Operators

- **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

Immediate Operands

- Small constants are used quite frequently (50% of operands)
- Make operand part of instruction itself!

- Design Principle 4: *Make the common case fast*

- *Example*: addi $sp, $sp, 4 # $sp = $sp + 4, $sp=29

001000	11101	11101	0000000000000100
6 bits	5 bits	5 bits	16 bits
op	rs	rt	16 bit number

- What If Constants are LARGER than 16-bits ?

How about larger constants?

- First we need to load a 32 bit constant into a register
- Must use two instructions for this: first new *load upper immediate* instruction for upper 16 bits

```
lui $t0, 1010101010101010
```

How about larger constants?

- To load $t0 with 1010… upto 32 bits
- `lui $t0, 1010101010101010`

filled with zeros

1010101010101010	0000000000000000

- Then get lower 16 bits in place:

  `ori $t0, $t0, 1010101010101010`

	1010101010101010	0000000000000000
	0000000000000000	1010101010101010
ori	1010101010101010	1010101010101010

- Now the constant is in place, use register-register arithmetic

Larger Constants

- Example:

Load the register $s0 with the value:

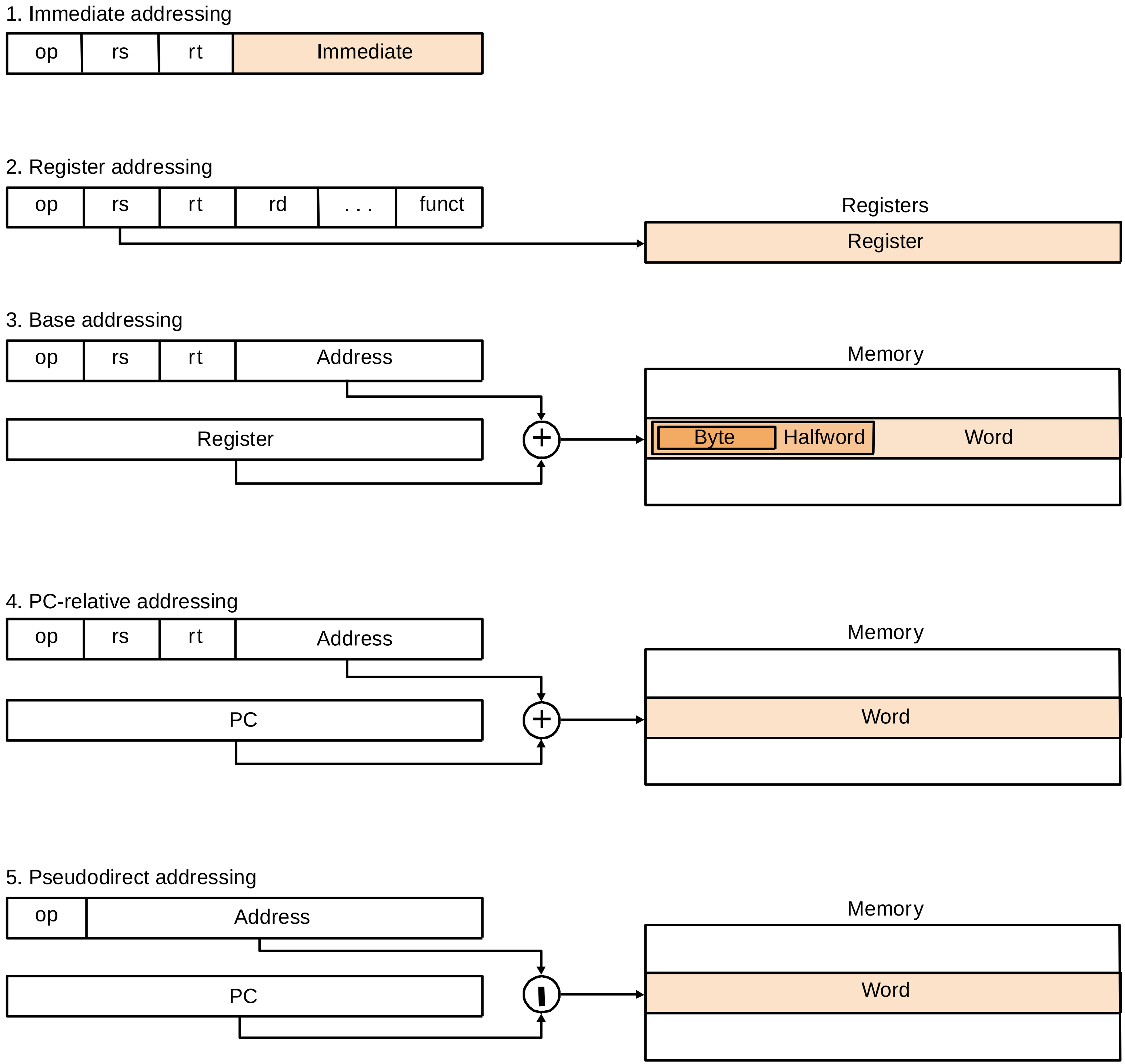
$4000000_{10} = 3D0900_{16}$

$= 0000\ 0000\ 0011\ 1101\ 0000\ 1001\ 0000\ 0000_2$

- MIPS Code:

```
lui $s0 61₁₀                  # 61₁₀   = 0000 0000 0011 1101₂
addi $s0, $s0, 2304₁₀         # 2304₁₀ = 0000 1001 0000 0000₂
```

MIPS Addressing Modes



So far

- Instruction / Format / Meaning

Instruction	Format	Meaning
`add $s1,$s2,$s3`	R	`$s1 = $s2 + $s3`
`sub $s1,$s2,$s3`	R	`$s1 = $s2 - $s3`
`lw $s1,100($s2)`	I	`$s1 = Memory[$s2+100]`
`sw $s1,100($s2)`	I	`Memory[$s2+100] = $s1`
`bne $s4,$s5,Lab1`	I	`Next instr. is at Lab1 if $s4 != $s5`
`beq $s4,$s5,Lab2`	I	`Next instr. is at Lab2 if $s4 = $s5`
`j Lab3`	J	`Next instr. is at Lab3`

- Formats:
- Simple instructions – all 32 bits wide, Very structured – no unnecessary baggage, Only three instruction formats

<table>
<tr><td>R</td><td>op</td><td>rs</td><td>rt</td><td>rd</td><td>shamt</td><td>funct</td></tr>
<tr><td>I</td><td>op</td><td>rs</td><td>rt</td><td colspan="3">16 bit address</td></tr>
<tr><td>J</td><td>op</td><td colspan="5">26 bit address</td></tr>
</table>

Summarize MIPS:

MIPS operands

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

Summary:
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Keep the register fields in the same place
    - Opcode always the first 6 bits

Summary:
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Keep the register fields in the same place
    - Opcode always the first 6 bits
- Smaller is faster
  - Limited instruction set
  - Limited number of registers in register file
  - Limited number of addressing modes

Summary: MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Keep the register fields in the same place
    - Opcode always the first 6 bits
- Smaller is faster
  - Limited instruction set
  - Limited number of registers in register file
  - Limited number of addressing modes
- Good design demands good compromises
  - Compromise between providing for larger addresses and constants in instructions
  - Keep all instructions of the same length

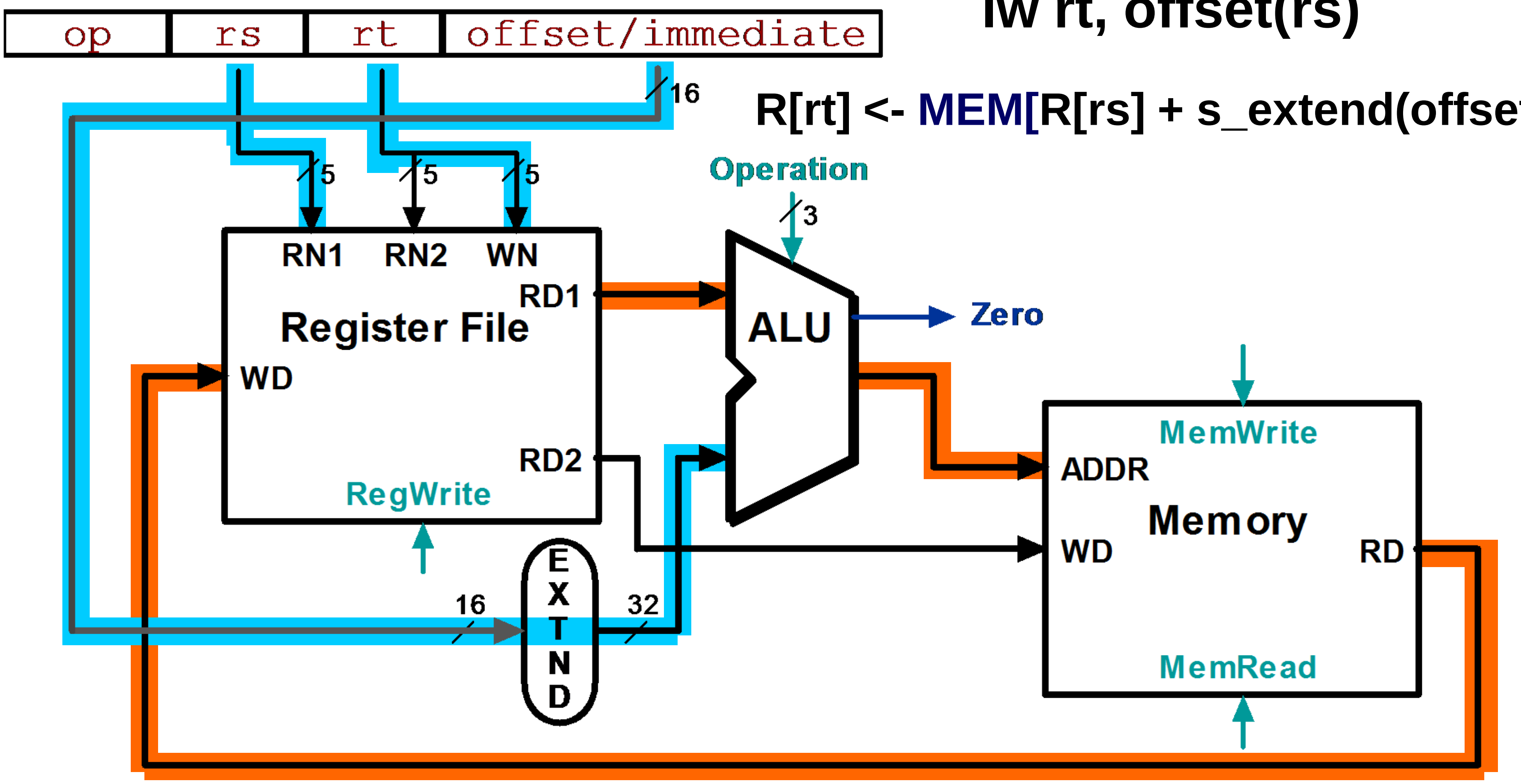
Summary:
MIPS (RISC) Design Principles

- Simplicity favors regularity
  - Fixed size instructions
  - Small number of instruction formats
  - Keep the register fields in the same place
    - Opcode always the first 6 bits
- Smaller is faster
  - Limited instruction set
  - Limited number of registers in register file
  - Limited number of addressing modes
- Good design demands good compromises
  - Compromise between providing for larger addresses and constants in instructions
  - Keep all instructions of the same length
- Make the common case fast
  - Arithmetic operands from the register file (load/store machine)
  - Allow instructions to contain immediate operands

Animating the Datapath

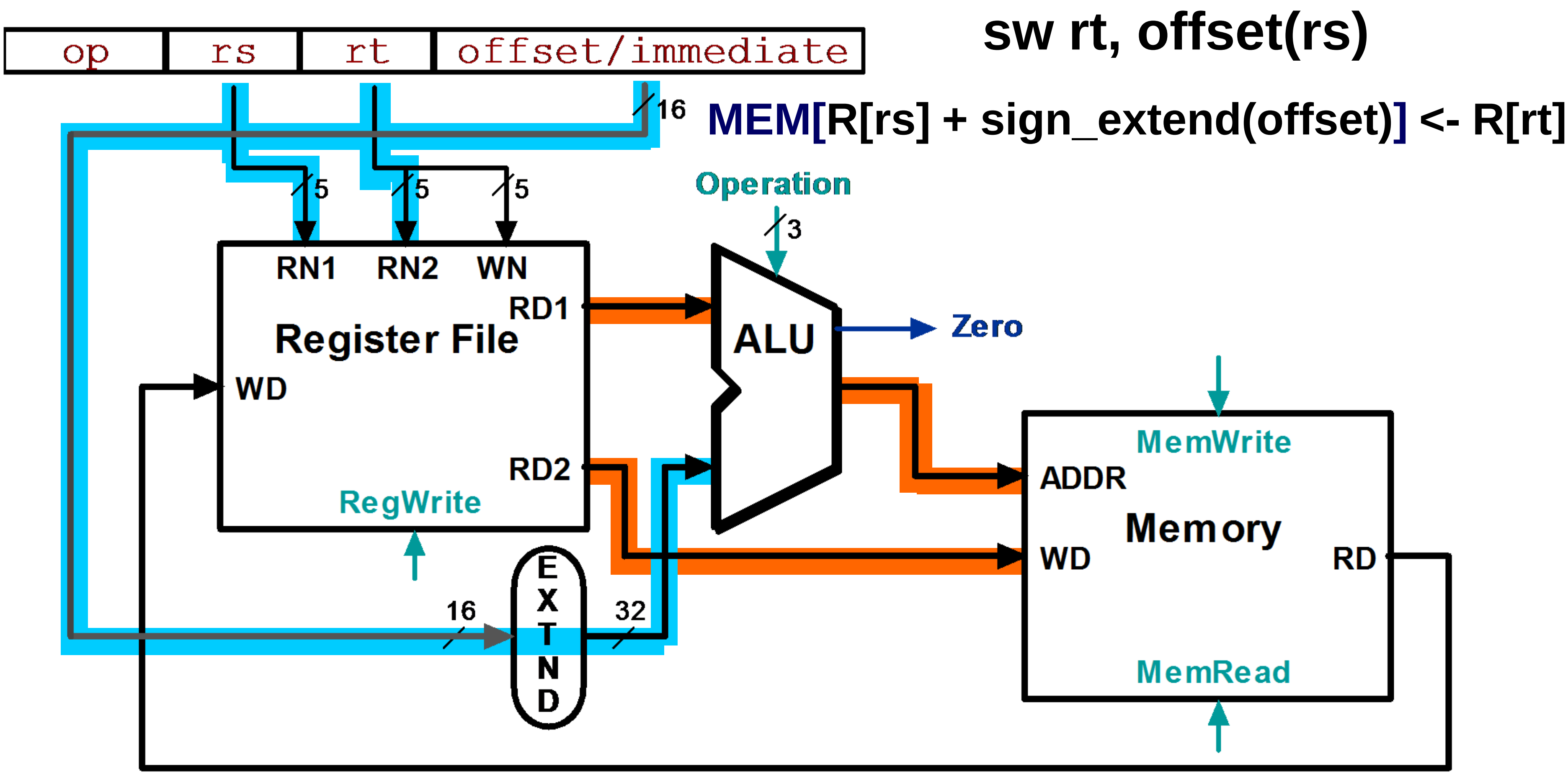
lw rt, offset(rs)

R[rt] <- MEM[R[rs] + s_extend(offset)];



Back

Animating the Datapath



- End of Presentation

- What is the role of the stack frame pointer? That is, what is it supposed to point to?
- The stack frame pointer is a register (%ebp) supposed to point to the first value in the stack frame for the currently-executing function
- The stack frame pointer is stored in the register %ebp, and must not be confused with the stack pointer, which is stored in the register %esp. They are different and play different roles.

- Why is it sometimes necessary to save the value of the stack frame pointer?
- When a function call occurs, stack frame pointer must be reset to point to the beginning of the stack frame for the called function; if we do not save the old value of stack frame pointer, there will be no way to reset stack frame pointer to the beginning of the caller's frame when the called function returns

- An x86-32 assembly procedure typically begins with the following two statements. Explain what each statement does.
- pushl %ebp # 1 #ebp stack frame pointer
- movl %esp, %ebp # 2
- The first statement stores a copy of the frame pointer on the stack (decrements %esp by 4; writes current value in %ebp to target of %esp)
- The second statement copies the value of the stack pointer %esp into the frame pointer %ebp, setting the frame pointer to point to the beginning of a new stack frame (and to the backed-up old value of the frame pointer).