EEL 4768: Computer Architecture

Instruction Set Architecture (ISA)

Instructor: Zakhia (Zak) Abichar

Department of Electrical Engineering and Computer Science
University of Central Florida

Source: Appendix A
"Computer Architecture, A Quantitative Approach", Hennessy and Patterson, 5th Edition

Instruction Set Principles

- Various approaches have been used in the architecture design
- The table (next slide) shows a sample of CPUs and their design approach

2

Machine	Number of general-purpose registers	Architectural style	Year
EDSAC	1	Accumulator	1949
IBM 701	1	Accumulator	1953
CDC 6600	8	Load-store	1963
IBM 360	16	Register-memory	1964
DEC PDP-8	1	Accumulator	1965
DEC PDP-11	8	Register-memory	1970
Intel 8008	1	Accumulator	1972
Motorola 6800	2	Accumulator	1974
DEC VAX	16	Register-memory, memory-memory	1977
Intel 8086	1	Extended accumulator	1978
Motorola 68000	16	Register-memory	1980
Intel 80386	8	Register-memory	1985
ARM	16	Load-store	1985
MIPS	32	Load-store	1985
HP PA-RISC	32	Load-store	1986
SPARC	32	Load-store	1987
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992
HP/Intel IA-64	128	Load-store	2001
AMD64 (EMT64)	16	Register-memory	2003

3

Instruction Set Architecture (ISA)

- The ISA consists of:
  - The instruction set (list of all instruction in the assembly language)
  - The instruction set includes the addressing mode
  - The format of the instructions
- Who has to deal with the ISA?
  - Assembly language programmer should know the assembly instructions (and their use) to write a program
  - The CPU hardware engineer should know the ISA since the CPU hardware implements all the assembly instructions
  - A compiler writer should know the ISA because a compiler translates a high-level code (eg: C or C++) into assembly code

4

ISA's Impact on the Computer

- The ISA has a significant impact on the overall computer performance
- Various computer markets have their own specific requirements; a good ISA meets the requirements of the computer in which it's used

Desktop computing (PCs)

- This segment emphasizes computing with integer and floating-point data types
- There is little regard for program size since a PC has a lot of memory

5

Servers

- Servers nowadays are mainly used for databases, file servers and web applications
- Floating-point performance is not important in these
- But integers and character strings performance is important

Personal mobile devices and embedded applications

- Energy consumption should be minimized
- In some embedded applications, the cost should be reduced since the product itself is inexpensive
- Code size should be reduced because less memory means there's less power consumption and less cost of computer
- It's optional to have a dedicated floating-point unit, therefore, the chip cost can be reduced

6

ISA's Impact on the Computer

- Practically, a certain instruction set can be used in all the three computer types: desktop, server, mobile & embedded
- The MIPS instruction set is used in all of these three computer types

7

Layered Instruction Set

- Intel's x86 instruction set provides backwards compatibility to earlier architectures
- This is important commercially since a computer owner can run the program they've bought for their older computer
- However, x86 is a CISC architecture, contains complex instructions and leads to a complex hardware design
- RISC architectures, on the other hand, use simple instructions that make the hardware simple
- How can Intel keep its x86 architecture to maintain backward compatibility and, at the same time, benefit from RISC's simple hardware design?

8

Layered Instruction Set

- In the usual way, the instruction set is implemented directly by the hardware
- However, Intel came up with a solution where the x86 instructions go through a hardware layer and are transformed into RISC-like instructions
- Potentially, a complex x86 instruction becomes one or more of the RISC-like instructions
- The CPU's hardware then runs the RISC-like simple instructions

- This approach was used by Intel (e.g. in the Pentium 4 CPU)

9

Types of Instruction Set Architectures (ISA)

- ISAs can be classified based on the type of internal storage that they use
- The internal storage can be: stack, accumulator or registers

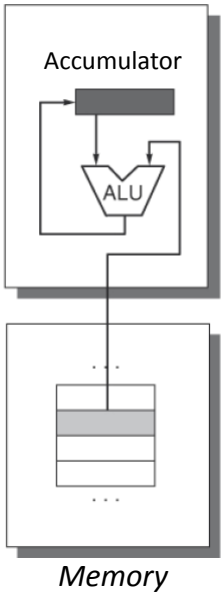
- The various types of ISAs are:
  - **Accumulator architecture**
  - **Stack architecture**
  - **Memory-memory architecture**
  - **Register-memory architecture**
  - **Load-store architecture**

10

Accumulator Architecture

- The accumulator architecture was the main approach among the earliest CPUs
- These CPUs didn't have a lot of storage space (not possible to put multiple registers)
- Therefore, the accumulator register was the default register that's used in all the operations

- The figure shows that the accumulator is always an operand of the ALU (called the implicit operand)
- The other operand is a data that's in the memory (called explicit operand)



11

Accumulator Architecture

- The code below evaluates the expression: **C = A+B**
- The variables, A, B and C are in the memory

- The first instruction loads the variable A in the accumulator (the label 'A' serves as a memory address)
- The second instruction grabs the variable B from the memory and adds it to the accumulator; the result of the addition goes in the accumulator
- The third instruction stores the accumulator in the memory at the variable C

```
Load    A       // load A in accumulator
Add     B       // add B to the accumulator
Store   C       // store the accumulator in C
```

12

Accumulator Architecture

- The accumulator code below evaluates this expression
- The variables A, B, C and D are in the memory

(A+B) * C / D

```
Load     A          // load A in the accumulator
Add      B          // add B to accumulator
Multiply C          // multiply C to the accumulator
Div      D          // divide the accumulator by D
```

- *What is the accumulator code for this expression?*

(A+B) / (C+D)

13

Accumulator Architecture

- One benefit to the accumulator architecture is that the instruction can be encoded on a small number of bits (like 8-bit instruction)

14

Accumulator Architecture

- Some accumulator architectures have more than one register
- But the accumulator is used in all the operations
- The other operand can be another register or a memory address

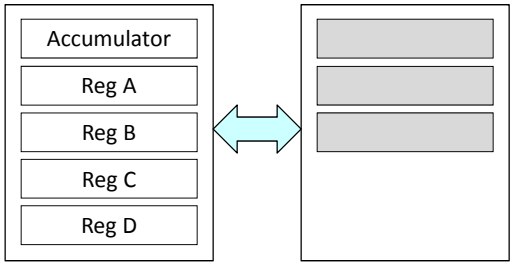
Instruction: **Add A**

- Accumulator = Accumulator + Register A

Instruction: **Add [200]**

- Accumulator = Accumulator +

Data at memory address 200



15

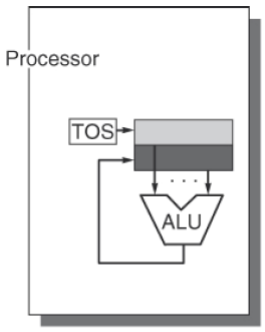
Accumulator Architecture

- The accumulator architecture is not used anymore in the modern CPU design
- The architectures that are popular today have general-purpose registers (GPR)
- A general-purpose register can be used to store any variable

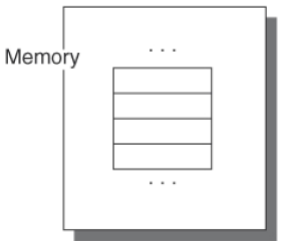
16

Stack Architecture

- The stack architecture has been used up to ~1980
- As the figure shows, the ALU operands are the two top locations of the stack
- (TOS: Top of Stack)



- The two operands (top 2 words on the stack) are popped from the stack, the ALU does the operation, and the result is pushed on the stack



- Data from memory can be loaded into the stack (at the top position)
- Similarly, the top position of the stack can be stored in the memory

17

Stack Architecture

- One benefit of the stack architecture is the compiler doesn't have to do variable-to-register allocation
- This used to be a difficult problem and the stack architecture tries to circumvent it

- There are two variants of the stack architecture:
- 1) The data and the operations are pushed on the stack
- 2) The data only is pushed on the stack; the operations come from the code

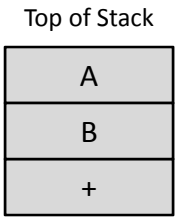
18

Stack Architecture

(Case 1) The data and the operations are pushed on the stack

- We want to evaluate this expression: **C = A + B**

- In this approach, the stack is initialized as below
- Then, the two operands are popped; the operation is popped; they're added and the results is stored on the stack

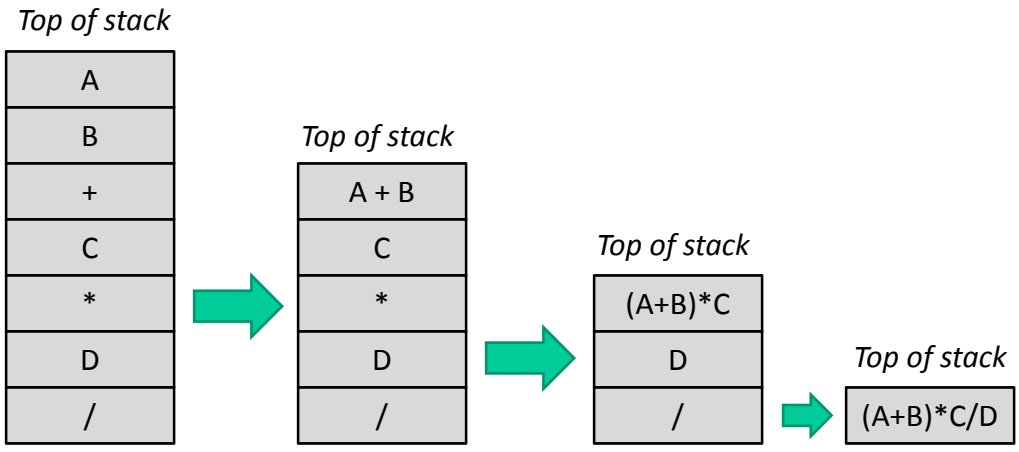


19

Stack Architecture

- This is the initialization of the stack to compute the expression:

(A+B) * C/D



20

Stack Architecture

(Case 2) Only the data goes on the stack

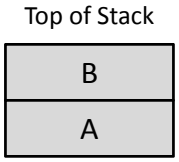
•This is the expression we're evaluating: **C = A + B**

•In this approach, the code pushes A and B from the memory onto the stack

•The 'Add' operation adds them

•Finally, the 'Pop' operation grabs the top of stack (the result of the addition) and stores it a the variable C in the memory

```
Push   A
Push   B
Add
Pop    C
```



21

Stack Architecture

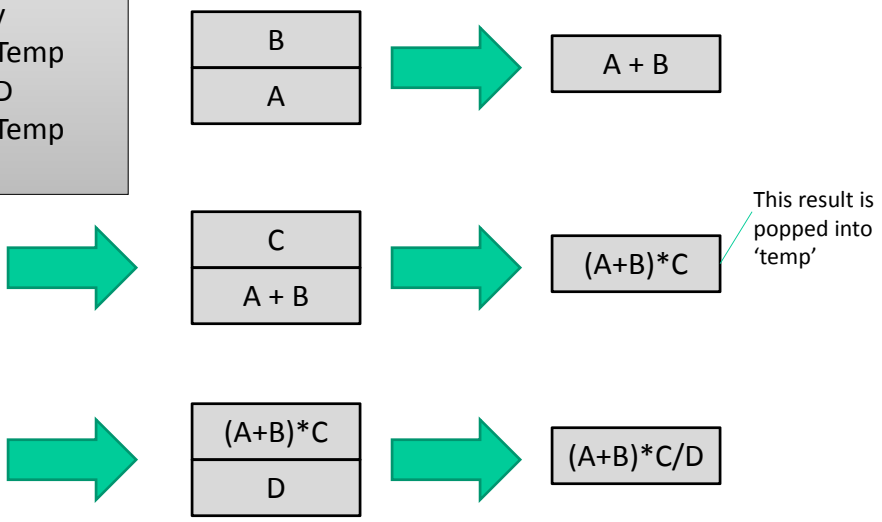
- The code below evaluates the expression: **(A+B) * C/D**
- *How can we rewrite this code without using the 'temp' variable?*

```
Push   A
Push   B
Add
Push   C
Multiply
Pop    Temp
Push   D
Push   Temp
Divide
```

22

```
Push   A
Push   B
Add
Push   C
Multiply
Pop    Temp
Push   D
Push   Temp
Divide
```

- This is the code and the corresponding operations



23

Stack Architecture

Where is the stack located?

- The stack is usually located in the memory
- However, since the stack is used extensively in the stack architecture, the top few words of the stack can be saved in registers to make the operations fast; the remaining stack locations are in the memory
- The stack architecture is not used in the modern CPU architecture
- One advantage for the stack architecture is that it's simple to write the compiler
- The problem of allocating the variables to register isn't difficult since all the variable are eventually passed to the top of the stack

24

Memory-Memory Architecture

- The memory-memory architecture keeps all the data in the memory (no data is stored in registers)
- This approach isn't used anymore in today's CPUs

- In the past this approach has been advocated since it makes the compiler simple
- The variables don't have to be allocated to registers since they always reside in the memory

25

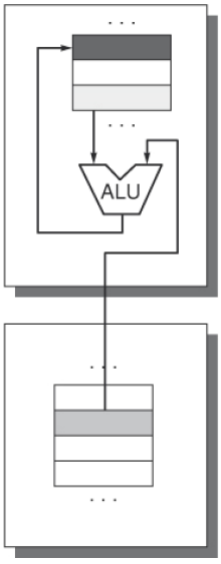
Register-Memory Architecture

- A register-memory architecture can operate on data that's in the memory directly

- As the figure shows, the ALU can have one operand as a register (top) and the other operand from the memory (bottom)
- Therefore, there's no need to load the data from memory into a register beforehand

- Some instructions use two registers

- However, usually, it's not possible for the ALU to use two memory locations (only one)



Memory

26

Register-Memory Architecture

- Register memory architecture typically use two operands in the instruction (add eax, ebx), as opposed to MIPS which uses three operands in the instruction (add t0, t1, t2)

- These are possible instruction in a register-memory architecture (from Intel x86)

```
ADD    EAX, EBX          # add two registers; leftmost one takes the
                         # result → EAX = EAX + EBX

ADD    EAX, [400]        # add register EAX and data @ address 400
                         # result goes in EAX
```

- However, it's not possible to add two memory locations in one instruction (there's at most one memory address)

27

Register-Memory Architecture

- Below is a code that evaluates the expression according to a register-memory architecture
- The variables A, B, C and D are located in the memory initially

(A + B) * C/D

```
Load    R1, A       // copy A from memory into R1
Add     R1, B       // add R1 to B from the memory
Mul     R1, C       // multiply R1 by C from the memory
Div     R1, D       // divide R1 by D from the memory
```

28

Register-Memory Architecture

- The advantage of a register-memory architecture is that data can be accessed in the memory directly (without having to do a separate instruction to copy the data into a register)
- A disadvantage of register-memory architectures is that some instructions take a few clock cycles (those who don't access the memory; however, other instructions take many more clock cycles (these instructions that access the memory)

29

Register-Memory Architecture

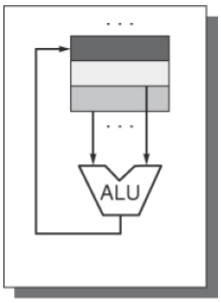
- A disadvantage of register-memory architectures is that instruction encoding is not uniform among all instructions
- Some instructions that don't have a memory address take a small number of bits, such as below

- On the other hand, instructions that have a memory address take a large number of bits, such as below

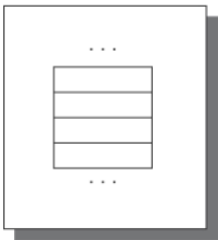
30

Load-Store Architecture

- This type of architecture is also called **'register-register architecture'**
- The two operands of the ALU are registers (an operand can't be a memory location)
- Therefore, the ALU can't access a memory location directly



- To do an ALU operation on a memory location, first, the data is copied from the memory into a register (an operation called 'load')
- Then, the ALU can access the data



Memory

31

Load-Store Architecture

- In a load-store architecture, the data is first 'loaded' from the memory into a register
- The data is then used in ALU computations
- Finally, the data is 'stored' back in the memory

- In load-store architectures, there are usually three operands per instruction
- This is the case in the MIPS architecture
- Example: **add $t0, $t1, $t2**
- The instruction above adds registers $t1 and $t2 and stores the result in register $t0 (registers $t1 and $t2 are not modified)

32

Load-Store Architecture

- The expression below is evaluated with the following code in a load-store architecture
- The variables A, B, C and D are initially in the memory

(A + B) * C/D

```
Load    R1, A          // copy A from memory into R1
Load    R2, B          // copy B from memory into R2
Add     R3, R1, R2     // R3 contains A+B
Load    R1, C          // copy C from memory into R1
Mul     R3, R3, R1     // R3 now contains (A+B)*C
Load    R1, D          // copy D from memory into R1
Div     R3, R3, R1     // R3 contains the final result
```

33

Load-Store Architecture

- One disadvantage of load-store architectures is that the number of instructions in the code is large
- This is because all the variables in the memory should be loaded first before they can be used
- *However, the register-memory can access memory data without loading them*

34

Load-Store Architecture

- The advantage of load-store architectures is that the instructions have a simple encoding and are usually fixed-length
- Below is an example of encoding:
- A register would take a few bits to encode since the number of registers in the CPU is small

- Also, since all the instruction take (more or less) the same number of cycles, they're more suitable to build the pipeline
- *However, register-memory instructions are not uniform in the instruction clock cycles and can't easily build an efficient pipeline*

35

Types of Instruction Set Architectures (ISA)

- Out of the multiple ISAs that we've seen, two are common nowadays: the register-memory architecture and the load-store architecture

Stack architecture	Not used anymore
Accumulator architecture	Not used anymore
Register-memory architecture	
Load-store architecture	
Memory-memory architecture	Not used anymore

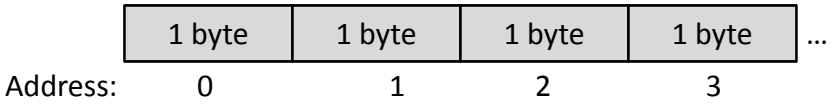
36

Memory Addressing

37

Memory Addressing

- A typical memory chip is ‘byte addressable’
- This means every byte has an address
- When the address increments by 1, we are referring to the next byte in the memory (as shown in the figure below)



- A byte is a small data type (can only represent 0 to 255 unsigned or from -128 to +127 signed)
- In a computer, a unit of data (word) is 16-bit, 32-bit or 64-bit
- Therefore, a ‘word’ occupies multiple memory addresses in the memory

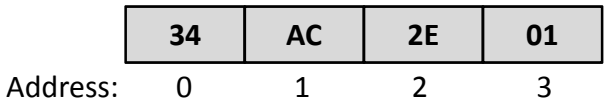
38

Little Endian vs. Big Endian

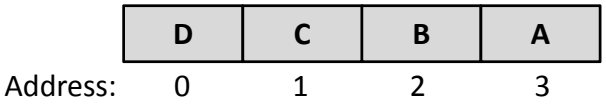
- Having a data type that's larger than one byte, there are two ways to store it in the memory

Little Endian

- In the Little Endian representation, the data type ends at the 'little address'
- Example1: The 32-bit hex number **01 2E AC 34** is represented in the memory as shown below



- Example2: The string "ABCD" is represented as shown below; this is a bit of a disadvantage since the string is spelled in reverse order in the memory

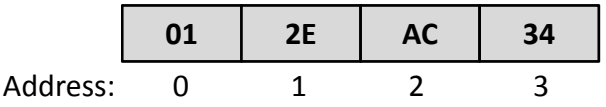


39

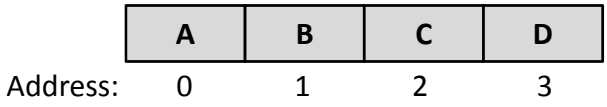
Little Endian vs. Big Endian

Big Endian

- In the Big Endian representation, the data type ends at the 'big address'
- Example1: The 32-bit hex number **01 2E AC 34** is represented in the memory as shown below



- Example2: The string "ABCD" is represented as shown below
- The Big Endian representation is preferred in this case since the string is stored in the same order it's read (when we're going up the addresses)



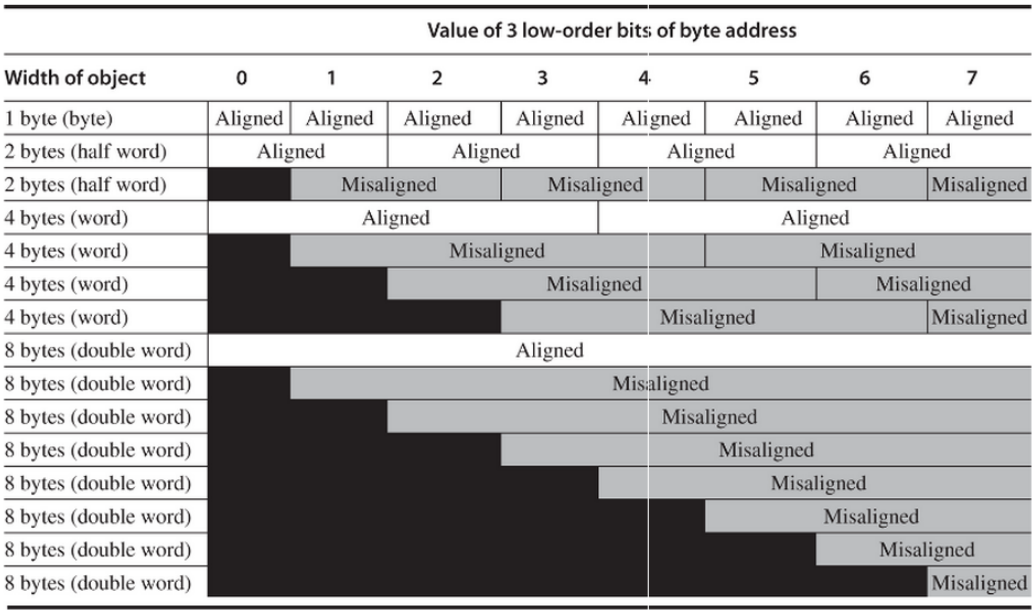
40

Memory Alignment

- The memory is aligned when access to data types that are larger than 1 byte happen at the natural boundaries
- In general, a data type of size 'n' bytes stored at adress 'A' is aligned if: **A modulo n = 0**
- Data types of 1 bytes are always aligned (see Figure A.5)
- Data types of 2 bytes are aligned when they're stored at bytes [0-1], [2-3], [4-5], …
- If a 2-byte data type is stored at [1-2] or [3-4], spanning the natural boundary, then it's not aligned

41

Figure A.5: Aligned and Misaligned Addresses



42

Memory Alignment

- Data type of 4 bytes are aligned when they're stored at bytes [0,1,2,3], [4,5,6,7], [8,9,10,11]…
- The lowest address is considered as the address so the data above is stored at addresses: 0, 4, 8, …
- This is the test equation:
  - 0 modulo 4 = 0
  - 4 modulo 4 = 0
  - 8 modulo 4 = 0

43

Memory Alignment

How is memory alignment relevant to performance?

- Figure A.5 shows a memory that reads out 8 bytes at a time
- Memory chips are typically aligned, which means the memory can provide one of these bytes in one read:
- Bytes [0-7] or [8-15] or [16-23] …
- If the data is stored at these boundaries, it will take only memory reference to read a word (of 8 bytes) from the memory
- If the data is stored in misaligned way, it will take two memory references to read a word (such as the last seven rows in Figure A.5)

44

Memory Alignment

- Memory alignment is also helpful in some aspects of the architecture
- For example, in MIPS, the memory is aligned and, therefore, the valid word (word=4 bytes) addresses are: **0, 4, 8, 12…**
- These addresses are multiples of 4 end in '00' when they're written in binary

- The encoding of the 'beq' (branch-on-equal) instruction uses this knowledge and the '00' are not encoded in the instruction
- They're added automatically when the instruction is decoded
- This increases the branch range by a factor of 4

45

Addressing Modes

- 'Addressing mode' refers to how an instruction specifies the address of its operands
- The address could be:
- (1) A register address that usually consists of a few bits
  - With 8 registers on the CPU, the register address is 3-bit, etc.
- (2) A memory address that usually has more bits than a register address
- (3) An immediate number
  - There's no real address here, the constant value is encoded in the instruction
- Figure A.6 shows the most popular addressing modes in CPUs

An addressing mode can be referred to by various names in different architectures.

46

Addressing mode	Example instruction	Meaning	When used
Register	`Add R4,R3`	`Regs[R4]←Regs[R4]+ Regs[R3]`	When a value is in a register.
Immediate	`Add R4,#3`	`Regs[R4]←Regs[R4]+3`	For constants.
Displacement	`Add R4,100(R1)`	`Regs[R4]←Regs[R4]+ Mem[100+Regs[R1]]`	Accessing local variables.
Register deferred or indirect	`Add R4,(R1)`	`Regs[R4]←Regs[R4]+ Mem[Regs[R1]]`	Accessing using a pointer or a computed address.
Indexed	`Add R3,(R1 + R2)`	`Regs[R3]←Regs[R3]+ Mem[Regs[R1]+Regs[R2]]`	Sometimes useful in array addressing: `R1` = base of array; `R2` = index amount.
Direct or absolute	`Add R1,(1001)`	`Regs[R1]←Regs[R1]+ Mem[1001]`	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect or memory deferred	`Add R1,@(R3)`	`Regs[R1]←Regs[R1]+ Mem[Mem[Regs[R3]]]`	If `R3` is the address of a pointer *p*, then mode yields **p*.
Autoincrement	`Add R1,(R2)+`	`Regs[R1]←Regs[R1]+ Mem[Regs[R2]]` `Regs[R2]←Regs[R2]+d`	Useful for stepping through arrays within a loop. `R2` points to start of array; each reference increments `R2` by size of an element, *d*.
Autodecrement	`Add R1,-(R2)`	`Regs[R2]←Regs[R2]-d` `Regs[R1]←Regs[R1]+ Mem[Regs[R2]]`	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	`Add R1,100(R2)[R3]`	`Regs[R1]← Regs[R1]+ Mem[100+Regs[R2]+Regs [R3]*d]`	Used to index arrays. May be applied to any indexed addressing mode in some machines.

Figure A.6

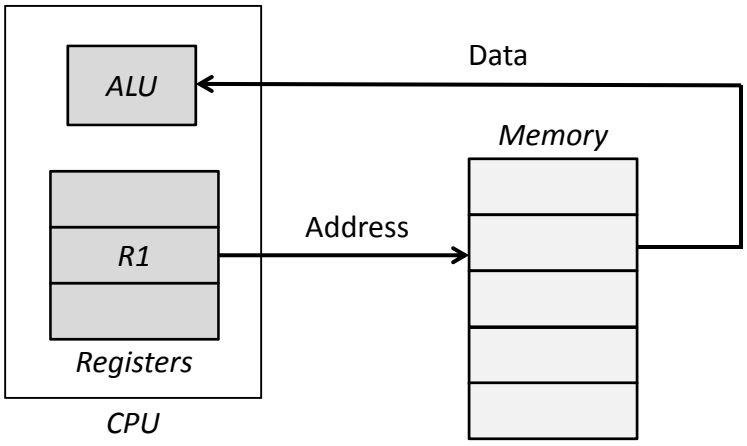
47

Addressing Modes

- Let's look at some of the addressing modes in Figure A.6

Register indirect Example: Add R4, (R1)

- In this addressing mode, register R1 is the address in the memory
- There is one memory access

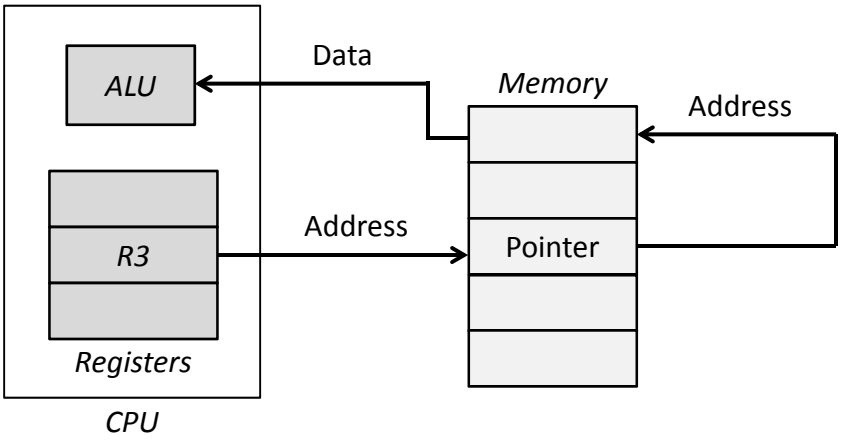


48

Addressing Modes

Memory indirect mode Example: Add R1, @(R3)

- Register R3 is the address of the pointer; once the pointer is read, another memory access fetches the data
- Here, there are two memory accesses

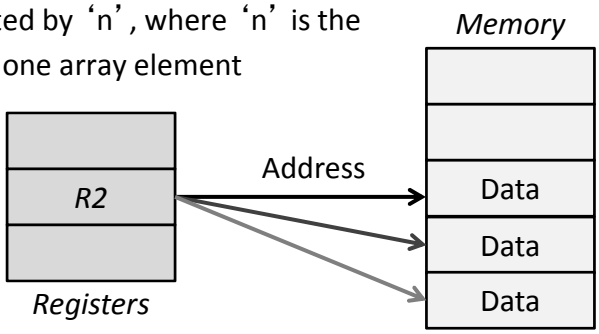


49

Addressing Modes

Autoincrement Example: Add R1, (R2)+

- This mode is used to access array elements in the memory
- R2 is used as the address of the data in memory
- When the data is fetched, R2 is incremented automatically so it's the address of the next array element
- Therefore, there is no need to increment R2 manually (through another 'add' instruction)
- R2 is incremented by 'n', where 'n' is the size (in bytes) of one array element



50

Addressing Modes

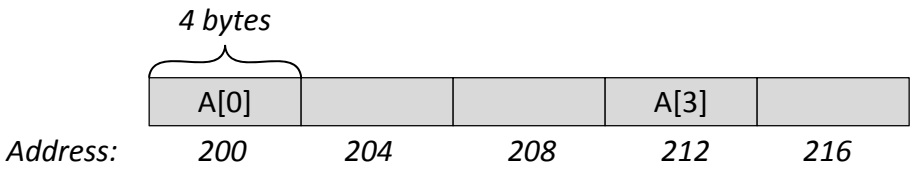
Scaled Example: Add R1, 100 (R2) [R3]

- This addressing mode is used to access data (array elements or data structures) from the memory
- First, let's look at the address of an array element in the memory

- The address of element A[y] is:

Start Address + (Element Size in bytes * y)

- For example, the address of A[3] = 200 + (4*3) = 212



51

Addressing Modes

Scaled Example: Add R1, 100 (R2) [R3]

- In this addressing mode, the address of the data in memory is:

100 + R2 + (R3 * scale)

- The scale is the size of the array element
  - If the array element is 4 bytes, then scale=4

- Let's consider this instruction: Add R1, 0 (R2) [R3]
- We initialize: R2=400 (the start address of the array) and R3=3 (since we want to access Array[3])
- The address is: 0 + R2 + (4*R3) = 400 + 4*3 = 412
  - This is the address of Array[3] in the memory)

52

Addressing Modes

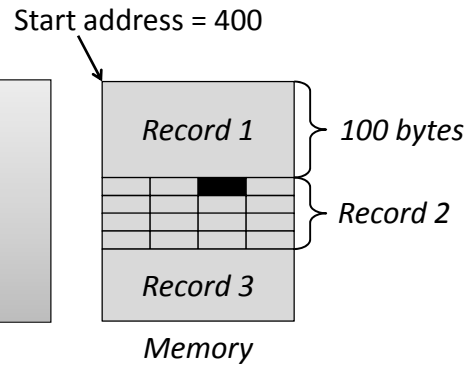
Scaled Example: Add R1, 100 (R2) [R3]

- Why does the scaled addressing mode contain a constant number?
  - The number '100' in the instruction above
- The constant number is used to skip a 'record' in a data structure
- The figure below shows a data structure that consists of multiple records

What's the address of the 'black box' element?

It's the element A[2] of Record 2

400 + 100 + 2*4 = 508



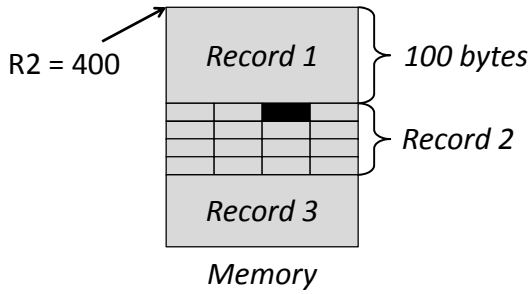
53

Addressing Modes

- Let's access the 'black box' element in the figure; this element is index 2 of Record 2
- We initialize: R2=400 (the start address of the data structure in memory) and R3=2 (the index of the element we want to access)

- We use this instruction: **Add R1, 100 (R2) [R3]**
- The address of the data accessed is: 100 + R2 + (4*R3)

- The value '100' is used to skip Over Record 1
- R3 is multiplied by 4 since every element is 4 bytes



54

Addressing Modes

- Another addressing mode that's not listed in Figure A.6 is **'PC-Relative Addressing'**
- The PC-relative addressing mode is used in 'branch' instructions
- Example of syntax: **branch R1, R2, Label**
- This is a possible encoding:

Opcode	*R1*	*R2*	Offset

- In PC-relating addressing mode, the 'offset' is added to the PC (Program Counter)
- Branch address is: **PC + Offset**

55

Addressing Modes: Simple vs. Complex

Simple Addressing Modes	Somewhere in the middle	Complex Addressing Modes
• Register `Add R4, R3`	• Displacement `Add R4, 100(R1)`	• Memory indirect `Add R1,@(R3)`
• Immediate `Add R4, #3`	• Register indirect `Add R4,(R1)`	• Autoincrement `Add R1,(R2)+`
• Direct `Add R1, (1001)`	• Indexed `Add R3,(R1+R2)`	• Autodecrement `Add R1,-(R2)`
	• PC-Relative `Address = PC+Offset`	• Scaled `Add R1,100(R2)[R3]`

56

Addressing Modes: Simple vs. Complex

- What are the pros and cons to simple and complex addressing modes?

Simple Addressing Modes	
Advantage	• Keep the hardware simple because the hardware implements the instructions • Keep the CPI (Clocks-per-instruction) small since the instruction does a small task
Disadvantage	• More instructions will be used because there's less flexibility in accessing data from the memory (additional instructions are used to compute memory addresses)

57

Addressing Modes: Simple vs. Complex

Complex Addressing Modes	
Advantage	• Reduce the instruction count since the instruction is 'powerful' in its ability to access data from the memory; this reduces the memory use
Disadvantage	• The hardware is complex since it implements the instructions' complex ways of access the memory • There will be a great variations between the number of clock cycles used by the instructions (instructions that use simple modes need few clock cycles; others that use the complex modes take more clock cycles); this variation makes it difficult to apply pipelining

58

The Popular Addressing Modes

If we're designing a new CPU, which addressing modes should we include?

- One way to find out is to measure the frequency of addressing modes in a typical program
- Our new architecture would adopt the addressing modes with the higher frequency since we can assume they are the most useful

- In this experiment, who will pick the addressing mode?
- The compiler! So this experiment assumes that the compiler is excellent and it's selecting the addressing modes that maximize the performance
- (if it's a bad compiler, the study will produce the wrong results)

- What kind of program should we choose?
- A program that's neutral and doesn't necessarily lead the compiler to favor one particular addressing mode (such as a benchmark)

59

The Popular Addressing Modes

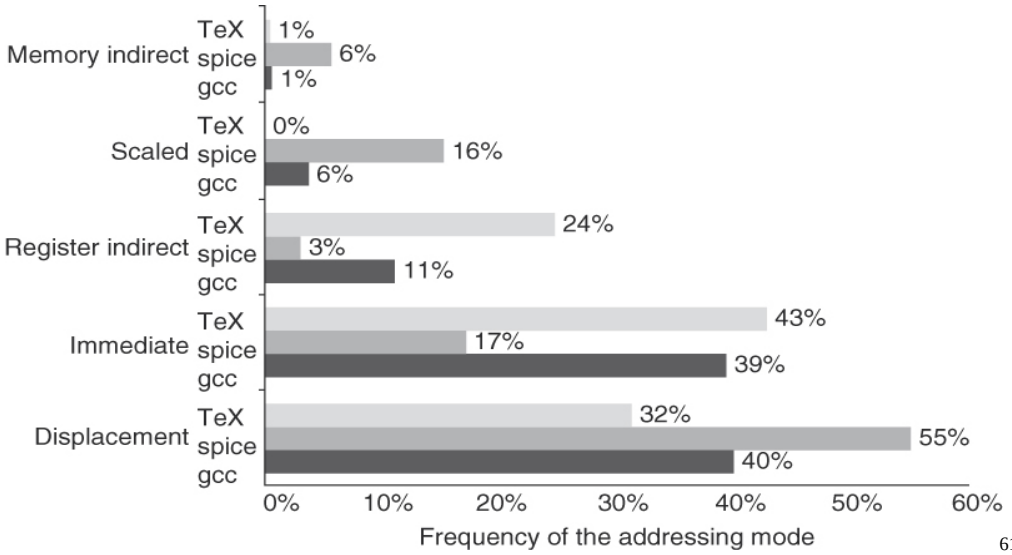
- What computer should we use for this study?
- A computer that has a lot of addressing modes since we want to see which ones the compiler will choose; so we better have a lot of addressing modes available so we can filter the good ones
- The study will use the VAX architecture since it supports a large number of addressing modes

Recap on the experiment setup
- The program is a fair program (benchmark)
- The computer supports a lot of addressing mode
- The compiler is unbiased; only aiming at high performance
- Due to these constraints, this study is considered a best attempt to get unbiased measurements as to which addressing mode is the most popular

60

The Popular Addressing Modes

- Three programs were run on the VAX computer: TeX, spice, gcc
- It turns out that the **'immediate'** and **'displacement'** addressing modes are the most used ones



61

The Popular Addressing Modes

- The two most popular addressing modes are "displacement" and "immediate"

- This immediate addressing mode contains a constant
- For example: addi t0, zero, 4
- It's useful since it's used to load constants in a register

- This is the displacement: lw t0, 12(s0)
- It's useful since it allows having an offset from the base register
- Therefore it can be used to access array elements
- A[4]=0; → sw zero, 12(s0)

62

Displacement Mode's Displacement Field

- *The displacement mode turned out to be a popular choice; let's look at the number of bits used for the displacement*
- This instruction uses displacement mode:

Add R4, 100 (R1)

- This is a possible encoding:

Opcode	*R1*	*R4*	Displacement

- How many bits should the 'displacement' field be?
- Figure A.8 shows the measurements of the number of bits that's used in the displacement field

63

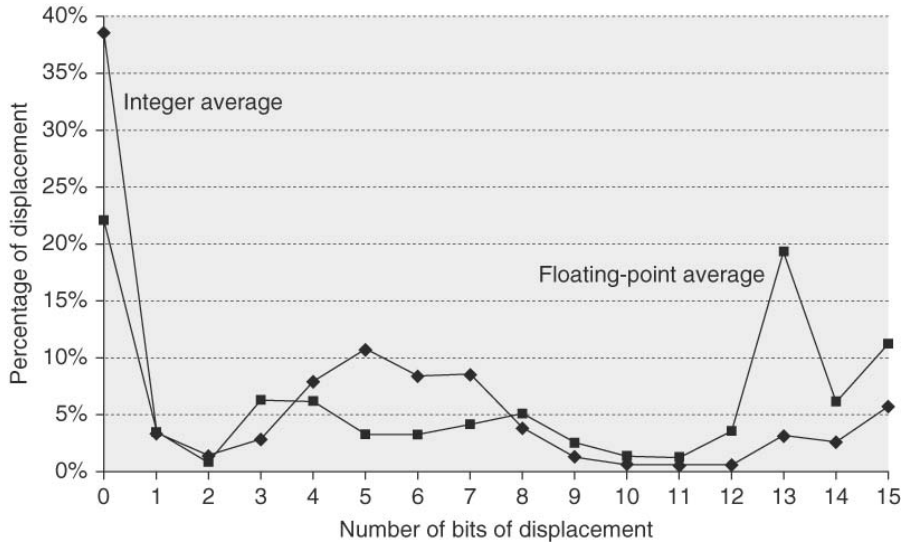


Figure A.8

- Data is measured on a load-store architecture; displacement field is 16 bits
- A high percentage of displacement values is 'zero'
- All the other values are used as well since the displacements vary based on how far the data is from the base register
- Figure shows that a 16-bit field is well utilized and is not an oversize choice

64

Immediate Field

- The immediate field represents a constant number that's encoded in the instruction
- **Terminology: Displacement vs. Immediate**
  - If the constant number is part of a memory address, it's called 'displacement'
  - If the constant number is loaded into a register or used in an arithmetic or logic operation, it's called 'immediate' field
- The constant number is a displacement:

Add R4, 100 (R1) // 100 is part of a memory address

- The constant number is an immediate:

Load R1, 200 // 200 is loaded in register R1 (no memory address)
Add R1, 300 // 300 is added to register R1 (no memory address)

65

Immediate Field

- How often are instructions with 'immediate' field used?
- Figure A.9 shows that immediates are used in 21% of integer instructions and 16% of floating-point instructions; they're quite useful

- 22% of the 'loads' load an immediate into a register (eg: Load R1, 200)
- The remaining load from the memory

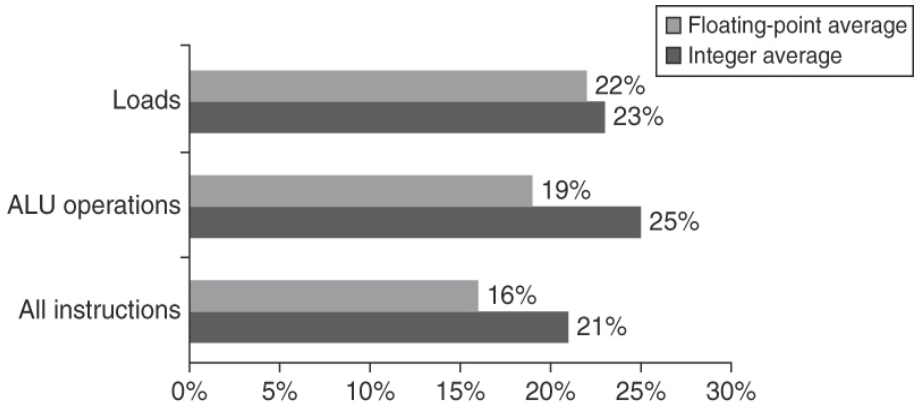


Figure A.9

66

Immediate Field

- How many bits should the 'immediate' field be?
- Figure A.10 shows measurements taken on the Alpha computer; the immediate field supported is 16-bit
- Figure A.10 shows that, for integer instructions, small immediate values are quite useful (that use 6 bits or less) and large immediate values are useful (that use 13 bits or more)
  - The values in the middle, that use between 7 and 12 bits are not as frequent
- Why are the small and large values more useful than the intermediate ones?

67

Immediate Fields

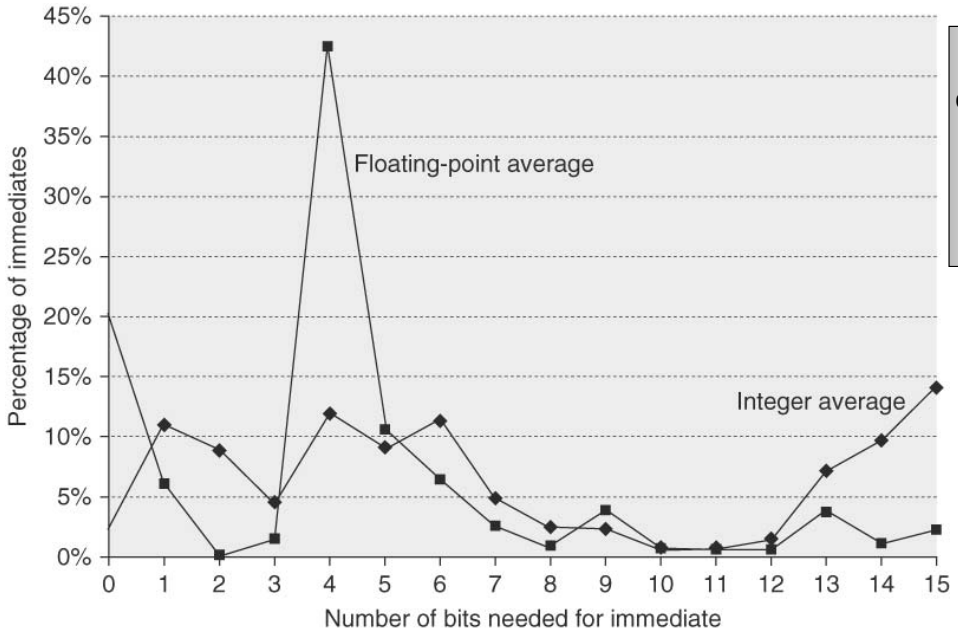


Figure A.10

68

Immediate Field

Small immediate values:

•Small immediate values are used in computations in a program

•For example, loading a small value as a loop counter bound; or incrementing a counter by 1

Large immediate values:

•The large immediate values occur when an address or a mask is loaded into a register (eg: load the address 8000 into R1)

•Another use of large immediate values is to load a 32-bit address into a register; this is done with two instructions such as below:

```
LUI   R1, 1000000100001111    // load upper immediate
ORI   R1, 0001111111000000    // load the rightmost 16 bits
```

69

Immediate Field

- Other measurements on the immediate field size were done on the VAX computer; the immediate field is 32 bits
- The results show that 20% to 25% of the immediate values were larger than 16 bits
- Therefore, using a 16-bit field is acceptable since it captures 75% to 80% of the values needed
  - (the cases that need more than 16 bits would use multiple instructions)
- The results also show that using an 8-bit immediate field captures 50% of the cases

70

Conclusion on Addressing Modes

- A newly designed CPU should support at least **'immediate', 'displacement'** and **'register indirect'** addressing modes due to their popularity
- The size of the 'displacement' field in the displacement addressing mode should be at least 12 to 16 bits
- The size of the 'immediate' field should be at least 8 to 16 bits

71

Type and Size of Operands

72

Type and Size of Operands

- The CPU should be able to do operations on multiple types and sizes of operands

Types: character, integer (arithmetic, logical), floating point

Size: 8-bit, 16-bit, 32-bit, 64-bit

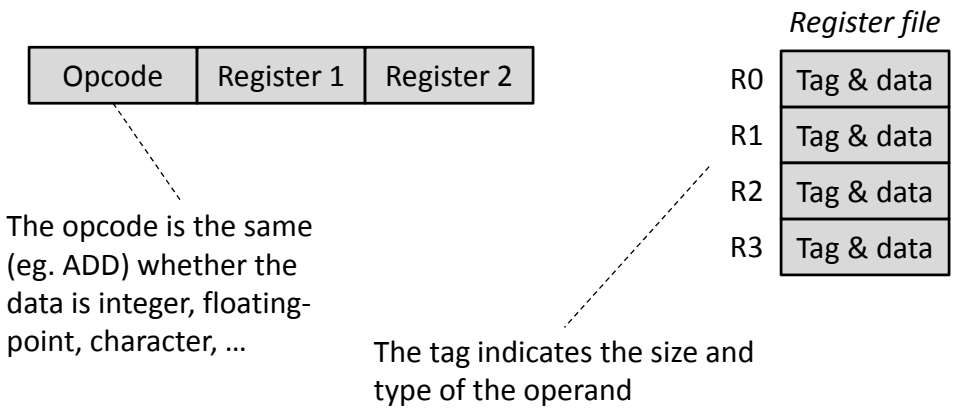
- How does the CPU know which type and size the operand is?
- The approach used nowadays is through the opcode (for every data type and size, a different opcode is used)

- Another approach used in earlier computers is called **'tagging'**

73

Type and Size of Operands

- With the tagging approach, the opcode is the same



- The tag approach is not used anymore
- Now we have clearly defined data types that are used across all computers

74

Type and Size of Operands

- These are the common data types used today
- For desktops and servers, the common data sizes are:
  - Character (8-bit)
  - Half word (16-bit)
  - Word (32-bit)
  - Single-precision floating-point (32-bit)
  - Double-precision floating-point (64-bit)
- The data type and size are often related
  - For example, a character is usually either 8-bit (ASCII) or 16-bit (Unicode)

75

Type and Size of Operands

- Nowadays, integers are represented as two's complement binary numbers on all computers
  - This wasn't the case in some of the earliest computers (sign-and-magnitude was used)
- Characters are ASCII, but nowadays, the 16-bit Unicode is becoming more popular since it supports a wide variety of languages
  - With a 16-bit character, we can represent 2^{16} = 65,536 unique characters
  - The Java programming language uses 16-bit Unicode characters

76

Type and Size of Operands

- The floating-point data type used to be specific to every CPU up until the mid 1980s
  - Since then, almost all computers started to use the IEEE 754 Standard to represent floating-point numbers
- Some architectures provide operations on characters to support string operations
  - A 'compare' instruction indicates if two characters are the same
  - A 'move' instruction copies a character from one location to another

77

Type and Size of Operands

- For business applications, some architectures support a decimal format called **'packed decimal'** or **'binary-coded decimal'**
- The digits are: 0 to 9
- Each digit is represented on a 4-bit number
- In addition, two digits are stored side-by-side in an 8-bit number
- This operation is called **'packing'**
- For example: the number 47 is written as 4: '0100' and 7: '0111'
- The packed representation is to put these two digits in one byte: '01000111'

78

Type and Size of Operands

- In the architectures that use packed decimals, a regular numeric string is called an **'unpacked decimal'** since each number takes one byte
- For example, the string "47" takes two bytes which are two ASCII codes
- "47" → "ASCII of 4", "ASCII of 7" → 00110100 00110111

- Some CPUs support packing and unpacking to convert between the two formats

79

Type and Size of Operands

- *What is the benefit for using the decimal representation in the computer?*

- Some floating-point numbers don't have an exact representation in binary
- For example, 0.1 is written in binary as: 0.0001100110011…
  - It doesn't have an exact representation
  - The term '0011' repeats infinitely

- If the application is a financial transaction 0.1 (10 cents) could be rounded up or down; 1 cent may be lost or gained
- The decimal representation is precise in this case

80

Type and Size of Operands

Which data types are the most used in a program?

- To find out, we'll see the measurement results on a benchmark program
- We'll see which data types are **referenced from the memory** (therefore, the data types used more often are the ones that should be optimized)

- The SPEC benchmark (next slide) uses:
  - Byte (character)
  - Half-word (short integer)
  - Word (integer)
  - Double-word (long integer)
  - Floating-point types

81

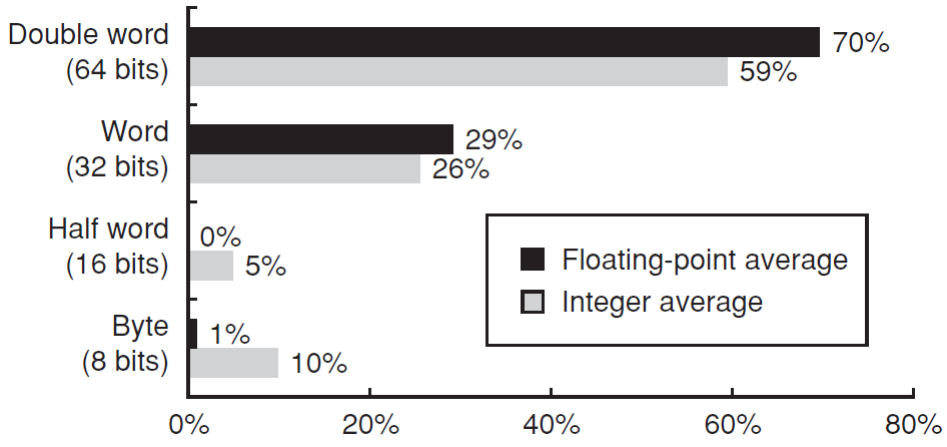


Figure A.11

- For integer operations, the 64-bit data type is the most popular
- For the floating-point operations, the 64-bit data type is also the most popular

- Otherwise, the 32-bit data (integer and floating-point) also have a 26-29% frequency of use

82

- Nowadays, an integer is usually a 64-bit number
- A signed 32-bit integer can represent numbers in the range of [-2billion, +2billion]; while this is suitable for many applications, some implementations prefer to use the 64-bit integer to reduce the posssibility of overflow
- For the floating-point, it's usually better to use a 'double' data type (rather than 'float') when a lot of precision digits are required; therefore, the 64-bit data type showed up as the most frequent

- 32-bit number, whether integer or floating-point, are still useful data types and showed a 26-29% use

- **Conclusion:** one conclusion we can draw from this result is that it makes sense to have a 64-bit path to the memory since most of the data fetched is 64-bit
- This will allow fetching the data from the memory in one clock cycle (versus 2 cycles if the path was 32-bit)

83

Type and Size of Operands

- Some architectures allow accessing objects in registers

- The VAX architecture allows accessing 8-bit or 16-bit from the register
- Practically, such accesses are infrequent
  - They account for 12% of register references
  - This is about 6% of all operand accesses

84

Operations in the Instruction Set

85

Operations in the Instruction Set

- The instruction set consists of a large number of instructions
- The instructions can be grouped in these categories:
  - Arithmetic and logical
  - Data transfer
  - Control (branches which can be conditional or unconditional)
  - …
- The table (next slide) shows the multiple categories

86

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiply, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

- The first three types **(arithmetic & logical, data transfer, control)** are usually well supported in all CPUs
- The **system** instructions vary widely among CPUs
- The last four categories **(floating-point, decimal, string, graphics)** range from no support at all to an extensive set of instructions depending on the CPU model

87

Operations in the Instruction Set

- The 'system' instructions provide functions such as switching the CPU between the 'user mode' and the 'kernel mode'
- Some of the basic CPU (such as in microcontrollers) usually run without an Operating System (OS) and therefore, might omit the system instructions

- The floating-point are sometimes part of an optional instruction set
- Some microcontroller CPUs don't have any floating-point instructions

- The decimal and string instructions are sometimes primitives (implemented in the hardware) or may be synthesized by the compiler from simpler instructions
- For example, if MIPS doesn't support 'load half' and 'load byte', we can still process 16-bit and 8-bit numbers, but it would take multiple instructions to do the task

88

Operations in the Instruction Set

- Graphics instructions typically operate on many small data types in parallel such as adding eight 8-bit numbers in parallel
- In this scenario, the ALU takes two 64-bit operands but each operand is interpreted as eight distinct 8-bit numbers

89

Operations in the Instruction Set

- Generally, across all architectures, **the most widely executed instructions** are the simple operations in the instruction set
- The table (next slide) shows 10 simple instructions that account for 96% of the instructions executed
  - The measurements are taken from a collection of integer programs running on an Intel 80x86 processor

90

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

- A computer designer would want to make these instructions run fast since they're executed frequently

91

Instructions for Control Flow

- Control flow instructions are the branch and the jump instructions in the code
  - Such instructions can be **'conditional'** or **'unconditional'**
- In the past (1950s), the control flow instructions were called **'transfers'**
- In the 1960s, the name **'branch'** started to be used
- In the book:
  - Branch refers to conditional instructions
  - Jump refers to unconditional instructions

Some architectures, like Intel x86, call everything 'jump', whether the instruction is conditional or unconditional.

92

Instructions for Control Flow

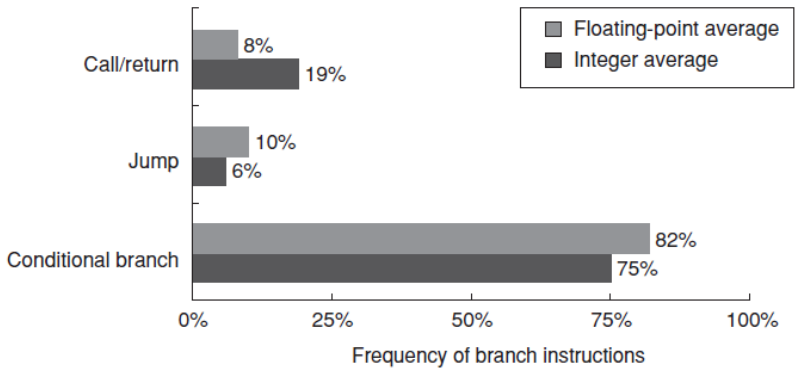
- There are the four different types of control flow instructions:
  - Conditional branches
  - Jumps
  - Procedure calls
  - Procedure returns

- The procedure call links the return address
- The procedure return jumps back to the calling code (possibly the 'main' function)

93

Instructions for Control Flow

- The figure shows which control flow instructions are the most popular



- The measurements were taken from a benchmark running on a load-store architecture

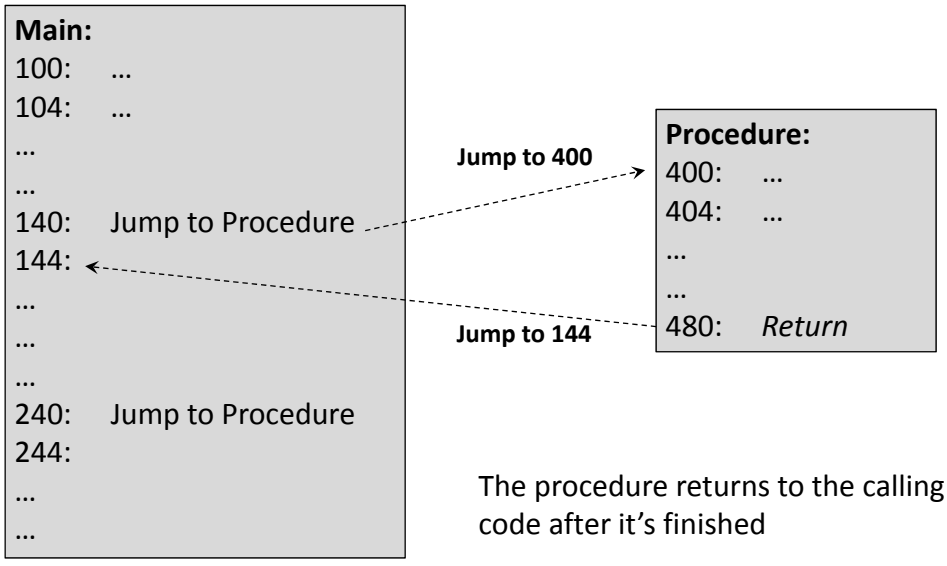
94

Instructions for Control Flow

- The destination address of a control flow instruction is usually specified in the instruction
- The exception to this is the 'procedure return' case
  - When the procedure returns, the program should go back to the calling code
  - The procedure could be called from multiple places in the code
  - Therefore, at compile time, the return address is not known

95

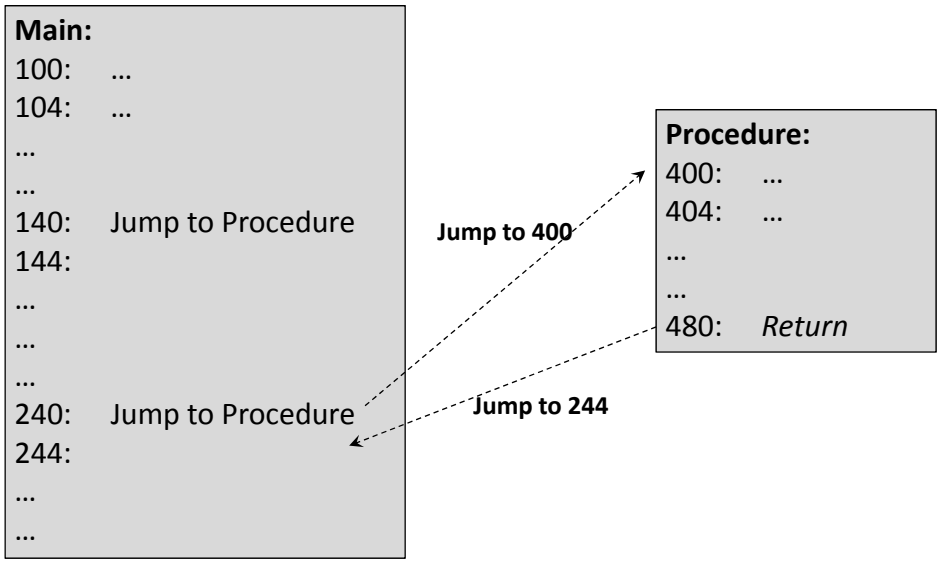
Instructions for Control Flow



The procedure returns to the calling code after it's finished

96

Instructions for Control Flow



97

Instructions for Control Flow

- The procedure returns to various addresses depending on where it was called from
- Therefore, the return address cannot be encoded in the jump instruction for procedure returns
- The return address is usually saved in a register and procedure returns use a jump-to-register instruction (eg: jr $ra), where $ra has the return address

98

Instructions for Control Flow

- The most common way to specify the destination is to use a displacement that's added to the PC
  - This is the **'PC-relative'** addressing mode
- PC-relative addressing is a good idea since the target address is often close to the PC
  - The PC-relative mode uses few bits to specify the offset between the PC and the target address
  - Rather than using a full address (32-bit or 64-bit) to indicate the branch target

99

Instructions for Control Flow

- The PC-relative mode allows the code to run independently of where it's loaded in the memory
  - This is referred to as **'position independence'**
- This eliminates some work when the program is linked
  - When the program is linked, library functions are added to the code written by the programmer
  - The jump addresses between the user code and the library functions are resolved
  - With PC-relative addressing, no addresses need to be resolved since the target address is specified relative to the PC
- The PC-relative mode is also suitable for programs that are linked dynamically at run-time

100

Instructions for Control Flow

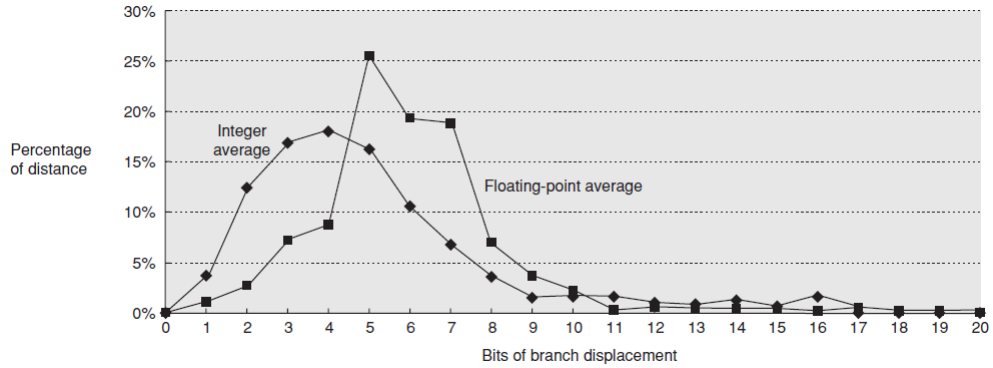
- To support 'procedure returns' the return address is specified at run time
  - Such an approach is called an **'indirect jump'**
  - The jump address is stored in a variable
  - An example would be: jump <reg>
- Some architectures allow the jump to use any addressing mode
  - Example: jump 400(reg) // address = reg + 400

101

Instructions for Control Flow

- Let's assume that branch instructions use the PC-relative mode
- *A relevant question is: how far is the branch address is from the branch instruction?*
- Knowing the branch distance allows us to determine the number of bits needed for the offset in the instruction
- The figure (next slide) shows the number of bits that's needed to encode the offset in a benchmark program

102



- Most of the branches that occur in the program span a small number of instructions
- Most of the cases can be encoded with an offset of 8 bits or less
- The measurements in the figure were taken on a load-store architecture (Alpha computer) with the instructions aligned on the word boundary

The branch offset is usually a signed number to allow jumping to a lower or higher address in the code. Practically, most of the executed branches branch to a higher address.

103

Conditional Branch Instructions

- Most of the control flow instructions executed are conditional branches
- *How is the condition mechanism implemented?*
- Three main approaches are used:
  - Condition code
  - Condition register
  - Compare and branch
- These approaches are summarized in the figure (next slide)

104

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Tests special bits set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

- In the **condition code approach,** there's a register called the 'condition code register' (CCR)
- It contains multiple fields, each field is 1 bit
- Example: CCR: [Z, S, …] (Z: zero bit, S: signed bit)
- The condition code register is set by some instructions that are done in the ALU
- For example, let's say the ADD instruction sets the condition code register

105

- When the ALU does an ADD operation, the condition code register is set automatically based on the result of the ADD
- If the result is zero, the Z bit becomes 1; otherwise, this bit is set to 0
- If the result is negative, the S bit becomes 1; otherwise it's set to 0

- A conditional branch is usually done via two instructions; the first instruction does an arithmetic (or logic) operation and sets the condition code register
- The following instruction inspects the condition code register and decides if the branch should be taken

- The code below branches to 'Label' if R1 is equal to R2
- There should not be any instruction between SUB and BRZ that changes the condition codes!

```
SUB   R3, R1, R2      //will do (R1–R2) and sets the flags
BRZ   Label           // will branch if Z flag =1 (which means
                      // R1 – R2=0)
```

106

Conditional Branch Instructions

- The Alpha architecture uses the **condition register approach**
- Its format of the branch instruction is:

Bxx <Reg> <Offset>

- The term 'Bxx' is replaced with 'BEQ' (branch on equal), 'BGT' (branch on greater than), etc.

- There is only one register in the branch instruction
- The other register is implied as the value zero

- Therefore, this instruction branches if R1 is equal to zero

BEQ R1, Label

107

Conditional Branch Instructions

- How can we use the 'condition code register' to branch if R1=R2?

```
SUB   R3, R1, R2        // will do: R3 = R1 – R2
BRZ   R3, Label         // will branch if R3=0 (which menas R1=R2)
```

108

Conditional Branch Instructions

- The Alpha architecture uses the value zero as the implied operand since in practice a lot of comparisons are done with zero

- Compilers also take advantage of this situation
- Let's say we write a loop in a high-level language that increments the counter from 0 to 9
- One way to check for loop termination is to do the subtraction (counter–10), if the result is zero, the loop stops
- *Therefore, the comparison is done with zero*

- The compiler might also change the loop to make the counter go from 9 down to 0 and compare the counter to 0 after the decrementation
- No extra subtraction is needed in this case (above, we did counter–10)
- *Here, also, the comparison is done with zero*

109

Conditional Branch Instructions

- Finally, some architectures do the **branch and comparison** all in one instruction
- The downside is the instruction needs a lot of computation

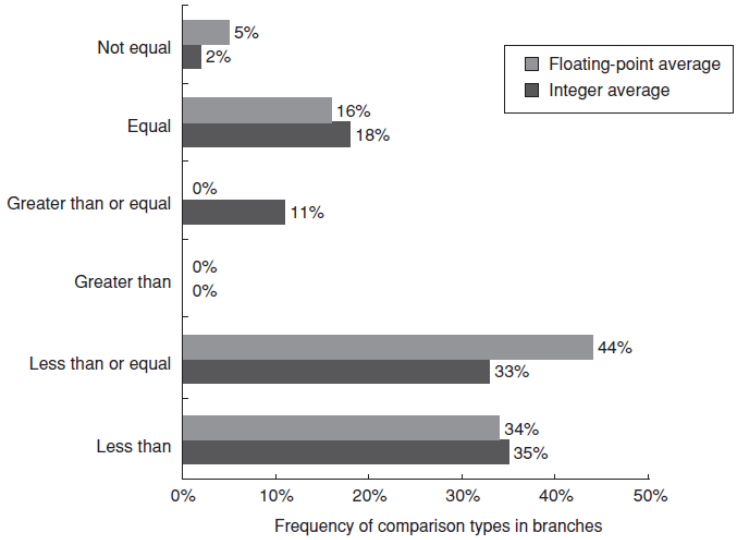
- In MIPS, the 'beq' and 'bne' instructions fall in this category

- For floating-point comparisons, MIPS uses the condition code approach
- Practically, the number of branches based on the floating-point comparisons is small
  - Branches are usually based on counters which are integers

110

Conditional Branch Instructions

- The figure shows the frequency of different comparisons used for conditional branches



111

Procedure Invocation Options

- The procedure calls and returns include 'control transfer' and 'state saving'
- Control transfer:
  - Branching to the procedure and returning to the calling code
- State saving
  - A procedure might alter a register that the calling code needs to use after the procedure returns
  - Such registers should be saved before the procedure is called; then, they should be restored when the procedure returns

112

Procedure Invocation Options

- When a procedure is called, one item that's always saved is the 'return address'
- This can be saved either in a special register or in a general-purpose register

MIPS:

- The instruction 'jal' (jump and link) jumps to the label and saves the return address automatically in register $ra

Jal Procedure1

  - The register $ra is always used, therefore, it's not encoded in the instruction
  - Such a register is generally called the **'link register'**

113

Procedure Invocation Options

Alpha:

•The Alpha architecture allows saving the return address in any register

BSR R1, Procedure

  - The instruction 'BSR' (Branch to SubRoutine) saves the return address in register R1
  - The register R1 is encoded in the instruction and any general-purpose register can be used

•In the case of MIPS and Alpha, a 'jump-to-register' instruction is used to return from the procedure (ex: jr $ra), (jr R1)

114

Procedure Invocation Options

- Two approaches have been used to save the state when a procedure is called:
  - Caller saving
  - Callee saving

Caller saving:

- The calling code saves the registers that it wants to use after the procedure returns
- The procedure code doesn't save any registers

115

Procedure Invocation Options

Callee saving:

- The calling code doesn't save the registers (it probably only saves the return address since the procedure can't find the return address)
- The procedure saves any register that it plans to use (they're usually saved in the memory at the stack)
- At the end of the procedure, the saved registers are restored before returning to the calling code

- ❖ Some older CPUs automatically save all the registers when a procedure is called
- ❖ This used to be a reasonable approach since the number of registers was small
- ❖ Nowadays, CPUs attempt to minimize the registers that should be saved to use fewer clock cycles in the procedure call

116

Procedure Invocation Options

- When the calling code and the procedure access the same global variable, 'caller saving' is preferred to 'callee saving'

- *Consider a global variable 'x'*
- Let's say the calling code has mapped 'x' to register R1
- If we do 'callee saving', the calling code keeps 'x' in R1
- This means the procedure has to know that 'x' is now mapped to R1 and use the value from there

- However, with 'caller saving', the calling code saves 'x' at a memory location before calling the procedure
- The procedure can get 'x' from the memory and use it
- This is the preferred approach since the procedure doesn't have to know the variable-to-register mapping of the calling code

117

Control Flow Instructions

Conclusion…

•Control flow instructions are used frequently

•The branch should be able to jump hundreds of instructions above or below the PC

•This means a PC-relative mode should have an offset of at least 8 bits

•The register indirect mode should be also supported (to enable procedure returns)

118

Summary so far…

The conclusion from this chapter so far…

- "We are leaning toward"
  - A load-store architecture
  - Addressing modes: displacement, immediate, register indirect
  - Integer data types are: 8-bit, 16-bit, 32-bit, 64-bit
  - Floating-point types are: 32-bit, 64-bit

- The instruction set includes:
  - Simple operations
  - PC-relative conditional branch
  - Jump-and-link instruction for procedure calling
  - Register indirect jump for procedure return

119