Computer Organization

Douglas Comer

Computer Science Department
Purdue University
250 N. University Street
West Lafayette, IN 47907-2066

http://www.cs.purdue.edu/people/comer

© Copyright 2006. All rights reserved. This document may not be reproduced by any means without written consent of the author.

V

Processor Types And Instruction Sets

What Instructions Should A Processor Offer?

- Minimum set is sufficient, but inconvenient
- Extremely large set is convenient, but inefficient
- Architect must consider additional factors
  - Physical size of processor
  - Expected use
  - Power consumption

The Point About Instruction Sets

The set of operations a processor provides represents a tradeoff among the cost of the hardware, the convenience for a programmer, and engineering considerations such as power consumption.

Representation

- Architect must choose
  - Set of instructions
  - Exact representation hardware uses for each instruction (*instruction format*)
  - Precise meaning when instruction executed
- Above items define the *instruction set*

Parts Of An Instruction

- Opcode specifies instruction to be performed
- Operands specify data values on which to operate
- Result location specifies where result will be placed

Instruction Format

- Instruction represented as binary string
- Typically
  - Opcode at beginning of instruction
  - Operands follow opcode

Illustration Of Typical Instruction Format



Instruction Length

- Fixed-length
  - Every instruction is same size
  - Hardware is less complex
  - Hardware can run faster
- Variable-length
  - Some instructions shorter than others
  - Appeals to programmers
  - More efficient use of memory

The Point About Fixed-Length Instructions

When a fixed-length instruction set is employed, some instructions contain extra fields that the hardware ignores. The unused fields should be viewed as part of a hardware optimization, not as an indication of a poor design.

General-Purpose Registers

- High-speed storage device
- Typically part of the processor
- Each register small size (typically, each register can accommodate an integer)
- Basic operations are *fetch* and *store*
- Numbered from 0 through N–1
- Many processors require operands for arithmetic operations to be placed in general-purpose registers

Floating Point Registers

- Usually separate from general-purpose registers
- Each holds one floating-point value
- Many processors require operands for floating point operations to be placed in floating point registers

Example Of Programming With Registers

- Add X and Y, and place result in Z
- Steps
  - Load a copy of X into register 3
  - Load a copy of Y into register 4
  - Add the value in register 3 to the value in register 4, and direct the result to register 5
  - Store a copy of the value in register 5 in Z
- Note: assumes registers 3, 4, and 5 are free

Terminology

- *Register spilling*
  - Refers to placing current contents of registers in memory for later recall
  - Occurs when registers needed for other computation
- Register allocation
  - Choose which values to keep in registers at any time
  - Programmer or compiler decides

Double Precision

- Refers to value that is twice as large as usual
- Hardware often uses a contiguous pair of registers to hold a double precision value

Register Banks

- Registers partitioned into disjoint sets called *banks*
- Additional hardware detail
- Optimizes performance
- Complicates programming

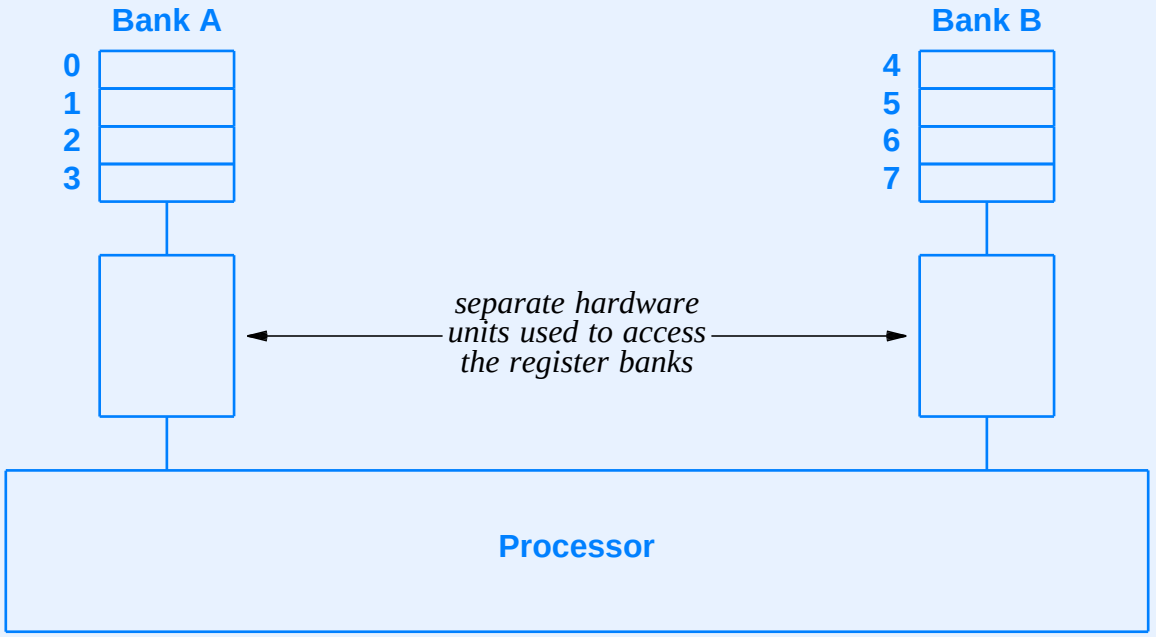
Register Banks

- Registers partitioned into disjoint sets called *banks*
- Additional hardware detail
- Optimizes performance
- Complicates programming

Typical Register Bank Scheme

- Registers divided into two banks
- ALU instruction that takes two operands must have one operand from each bank
- Programmer's responsibility

Illustration Of Register Banks



- Both banks can be accessed simultaneously

Consequence For Programmers

- Operands must be assigned to banks
- Even trivial programs cause problems
- Example

$$R \leftarrow X + Y$$
$$S \leftarrow Z - X$$
$$T \leftarrow Y + Z$$

Register Conflicts

- Occur when operands specify same register bank
- Reported by compiler / assembler
- Can require programmer to copy values to opposite register bank

Types Of Instruction Sets

- Two basic forms
  - Complex Instruction Set Computer (CISC)
  - Reduced Instruction Set Computer (RISC)

CISC Instruction Set

- Many instructions (often hundreds)
- Given instruction can require arbitrary time to compute
- Examples of CISC instructions
  - Move graphical item on bitmapped display
  - Memory copy or clear
  - Floating point computation

RISC Instruction Set

- Few instructions (typically 32 or 64)
- Each instruction executes in one clock cycle
- Example: MIPS instruction set

Summary Of Instruction Sets

A processor is classified as CISC if the instruction set contains instructions that perform complex computations that can require long times; a processor is classified as RISC if it contains a small number of instructions that can each execute in one clock cycle.

Execution Pipeline

- Hardware optimization technique
- Allows processor to complete instructions faster
- Typically used with RISC instruction set

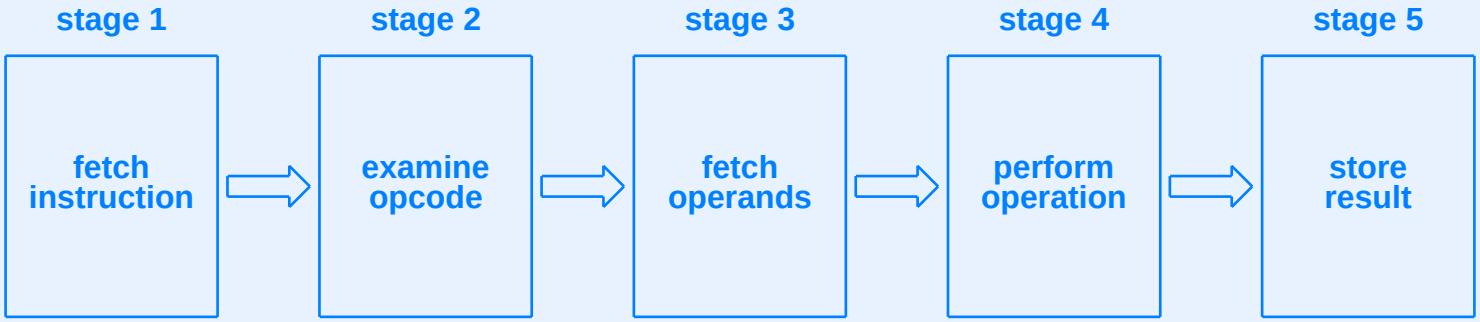
Typical Instruction Cycle

- Fetch the next instruction
- Examine the opcode to determine how many operands are needed
- Fetch each of the operands (e.g., extract values from registers)
- Perform the operation specified by the opcode
- Store the result in the location specified (e.g., a register)

To Optimize Instruction Cycle

- Build separate hardware block for each step
- Arrange to pass instruction through sequence of hardware blocks

Illustration Of Execution Pipeline



- Example pipeline has five stages

Pipeline Speed

- All stages operate in parallel
- Given stage can start to process a new instruction as soon as current instruction finishes
- Effect: N-stage pipeline can operate on N instructions simultaneously

Illustration Of Instructions In A Pipeline

Time

clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. 1	-	-	-	-
2	inst. 2	inst. 1	-	-	-
3	inst. 3	inst. 2	inst. 1	-	-
4	inst. 4	inst. 3	inst. 2	inst. 1	-
5	inst. 5	inst. 4	inst. 3	inst. 2	inst. 1
6	inst. 6	inst. 5	inst. 4	inst. 3	inst. 2
7	inst. 7	inst. 6	inst. 5	inst. 4	inst. 3
8	inst. 8	inst. 7	inst. 6	inst. 5	inst. 4

RISC Processors And Pipelines

Although a RISC processor cannot perform all steps of the fetch-execute cycle in a single clock cycle, an instruction pipeline with parallel hardware provides approximately the same performance: once the pipeline is full, one instruction completes on every clock cycle.

Using A Pipeline

- Pipeline is *transparent* to programmer
- Disadvantage: programmer who does not understand pipeline can produce inefficient code
- Reason: hardware automatically *stalls* pipeline if items are not available

Example Of Instruction Stalls

- Assume
  - Need to perform addition and subtraction operations
  - Operands and results in registers *A* through *E*
  - Code is:

Instruction K: C ← add A B
Instruction K+1: D ← subtract E C

- Second instruction stalls to wait for operand *C*

Effect Of Stall On Pipeline

Time

clock	stage 1	stage 2	stage 3	stage 4	stage 5
1	inst. K	inst. K-1	inst. K-2	inst. K-3	inst. K-4
2	inst. K+1	inst. K	inst. K-1	inst. K-2	inst. K-3
3	inst. K+2	inst. K+1	inst. K	inst. K-1	inst. K-2
4	inst. K+3	inst. K+2	(inst. K+1)	inst. K	inst. K-1
5	-	-	(inst. K+1)	-	inst. K
6	-	-	inst. K+1	-	-
7	inst. K+4	inst. K+3	inst. K+2	inst. K+1	-
8	inst. K+5	inst. K+4	inst. K+3	inst. K+2	inst. K+1

- *Bubble* passes through pipeline

Potential Causes Of A Pipeline Stall

- Access external storage
- Invoke a coprocessor
- Branch to a new location
- Call a subroutine

Achieving Maximum Speed

- Program must be written to accommodate instruction pipeline
- To minimize stalls
  - Avoid introducing unnecessary branches
  - Delay references to result register(s)

Example Of Avoiding Stalls

```
C ← add A B
D ← subtract E C
F ← add G H
J ← subtract I F
M ← add K L
P ← subtract M N
```

(a)

```
C ← add A B
F ← add G H
M ← add K L
D ← subtract E C
J ← subtract I F
P ← subtract M N
```

(b)

- Stalls eliminated by rearranging (a) to (b)

A Note About Pipelines

Although hardware that uses an instruction pipeline will not run at full speed unless programs are written to accommodate the pipeline, a programmer can choose to ignore pipelining and assume the hardware will automatically increase speed whenever possible.

No-Op Instructions

- Have no effect on
  - Registers
  - Memory
  - Program counter
  - Computation
- Can be inserted to avoid instruction stalls
- Often used by a compiler

Use Of No-OP

- Example

  Instruction K: C ← add A B

  Instruction L+1: no-op

  Instruction K+2: D ← subtract E C

- No-op allows time for result from register *C* to be fetched for *subtract* operation

Forwarding

- Hardware optimization to avoid stall
- Allows ALU to reference result in next instruction
- Example

Instruction K: C ← add A B

Instruction K+1: D ← subtract E C

- Forwarding hardware passes result of *add* operation directly to next instruction

Types Of Operations

- One possible categorization
  - Arithmetic instructions (integer arithmetic)
  - Logical instructions (also called Boolean)
  - Data access and transfer instructions
  - Conditional and unconditional branch instructions
  - Floating point instructions
  - Processor control instructions

Program Counter

- Hardware register
- Used during fetch-execute cycle
- Gives address of next instruction to execute
- Also known as *instruction pointer*

Fetch-Execute Algorithm Details

Assign the program counter an initial program address.
Repeat forever {

Fetch: access the next step of the program from the location given by the program counter.

Set an internal address register, A, to the address beyond the instruction that was just fetched.

Execute: Perform the step of the program.

Copy the contents of address register A to the program counter.

}

Branches And Fetch Execute

- Absolute branch
  - Typically named *jump*
  - Operand is an address
  - Assigns operand to internal register *A*
- Relative branch
  - Typically named *br*
  - Operand is a signed value
  - Operand is added to internal register *A*

Subroutine Call

- Jump subroutine (*jsr* instruction)
  - Similar to a *jump*
  - Saves value of internal register *A*
  - Replaces *A* with operand address
- Return from subroutine (*ret* instruction)
  - Retrieves value saved during *jsr*
  - Replaces *A* with saved value

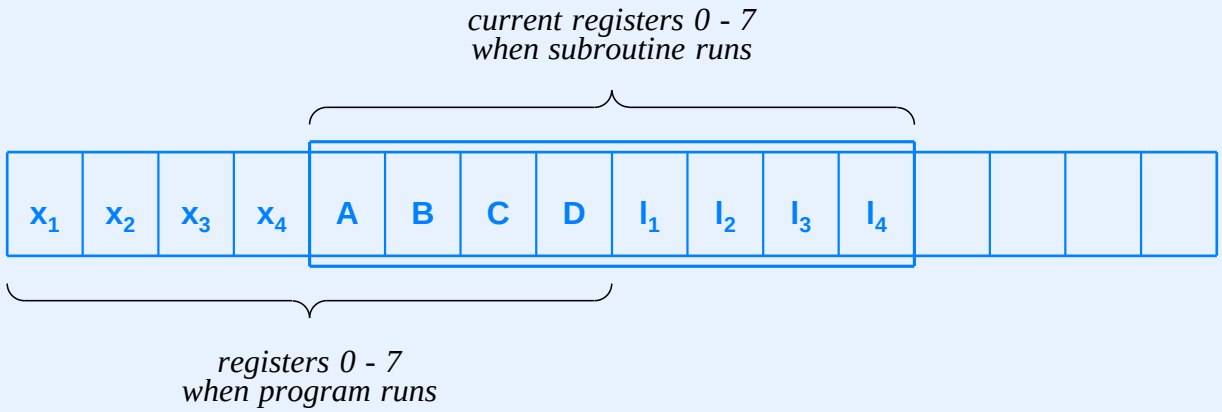
Passing Arguments

- Multiple methods have been used
- Examples
  - Store arguments in memory
  - Store arguments in special-purpose hardware registers
  - Store arguments in general-purpose registers
- Many techniques also used to return result from *function*

Register Window

- Hardware optimization for argument passing
- Processor contains many general-purpose registers
- Small subset of registers visible at any time
- Caller places arguments in reserved registers
- During procedure call, register window moves

Illustration Of Register Windows



Example Instruction Set

- Known as *MIPS* instruction set
- Early RISC design
- Minimalistic

MIPS Instruction Set (Part 1)

Instruction	Meaning
Arithmetic	
add	integer addition
subtract	integer subtraction
add immediate	integer addition (register + constant)
add unsigned	unsigned integer addition
subtract unsigned	unsigned integer subtraction
add immediate unsigned	unsigned addition with a constant
move from coprocessor	access coprocessor register
multiply	integer multiplication
multiply unsigned	unsigned integer multiplication
divide	integer division
divide unsigned	unsigned integer division
move from Hi	access high-order register
move from Lo	access low-order register
Logical (Boolean)	
and	logical *and* (two registers)
or	logical *or* (two registers)
and immediate	*and* of register and constant
or immediate	*or* of register and constant
shift left logical	Shift register left N bits
shift right logical	Shift register right N bits

MIPS Instruction Set (Part 2)

Instruction	Meaning
Data Transfer	
load word	load register from memory
store word	store register into memory
load upper immediate	place constant in upper sixteen bits of register
move from coproc. register	obtain a value from a coprocessor
Conditional Branch	
branch equal	branch if two registers equal
branch not equal	branch if two registers unequal
set on less than	compare two registers
set less than immediate	compare register and constant
set less than unsigned	compare unsigned registers
set less than immediate	compare unsigned register and constant
Unconditional Branch	
jump	go to target address
jump register	go to address in register
jump and link	procedure call

MIPS Floating Point Instructions

Instruction	Meaning
Arithmetic	
FP add	floating point addition
FP subtract	floating point subtraction
FP multiply	floating point multiplication
FP divide	floating point division
FP add double	double-precision addition
FP subtract double	double-precision subtraction
FP multiply double	double-precision multiplication
FP divide double	double-precision division
Data Transfer	
load word coprocessor	load value into FP register
store word coprocessor	store FP register to memory
Conditional Branch	
branch FP true	branch if FP condition is true
branch FP false	branch if FP condition is false
FP compare single	compare two FP registers
FP compare double	compare two double precision values

Aesthetic Aspects Of Instruction Sets

- Elegance
  - Balanced
  - No frivolous or useless instructions
- Orthogonality
  - No unnecessary duplication
  - No overlap among instructions

Principle Of Orthogonality

The principle of orthogonality specifies that each instruction should perform a unique task without duplicating or overlapping the functionality of other instructions.

Condition Codes

- Hardware bits
- Set by ALU
- Tested in conditional branch instruction

Example Of Condition Code

```
        cmp   r4, r5    # compare regs. 4 & 5, and set condition code
        be    lab1      # branch to lab1 if cond. code specifies equal
        mov   r3, 0     # place a zero in register 3
lab1:   …program continues at this point
```


Questions?