Computer Architecture

Prof. Dr. Nizamettin AYDIN

naydin@yildiz.edu.tr
nizamettinaydin@gmail.com

http://www.yildiz.edu.tr/~naydin

Computer Architecture

RISC Characteristics

Outline

Major Advances in Computers

- The family concept
  - IBM System/360 1964
  - DEC PDP-8
  - Separates architecture from implementation
- Microporgrammed control unit
  - Idea by Wilkes 1951
  - Produced by IBM S/360 1964
- Cache memory
  - IBM S/360 model 85 1969
- Solid State RAM
- Microprocessors
  - Intel 4004 1971
- Pipelining
  - Introduces parallelism into fetch execute cycle
- Multiple processors

RISC

- Reduced Instruction Set Computer (RISC)

- Key features
  - Large number of general purpose registers
  - And/or use of compiler technology to optimize register usage
  - Limited and simple instruction set
  - Emphasis on optimising the instruction pipeline

Comparison of processors

	Complex Instruction Set (CISC)Computer			Reduced Instruction Set (RISC) Computer		Superscalar		
Characteristic	IBM 370/168	VAX 11/780	Intel 80486	SPARC	MIPS R4000	PowerPC	Ultra SPARC	MIPS R10000
Year developed	1973	1978	1989	1987	1991	1993	1996	1996
Number of instructions	208	303	235	69	94	225		
Instruction size (bytes)	2–6	2–57	1–11	4	4	4	4	4
Addressing modes	4	22	11	1	1	2	1	1
Number of general-purpose registers	16	16	8	40 - 520	32	32	40 - 520	32
Control memory size (Kbits)	420	480	246	—	—	—	—	—
Cache size (KBytes)	64	64	8	32	128	16-32	32	64

Driving force for CISC

- Complex Instruction Set Computer (CISC)
- Software costs far exceed hardware costs
- Increasingly complex high level languages
- Major cost in the lifecycle of a system is software, not hardware
- Systems have also an element of unreliability
  - It is common for programs, both system and application, to continue to exhibit new bugs after years of operation
- Response from researchers and industry has been to develop ever more powerful and complex high-level programming languages
  - HLLs allow programmers to express algorithms more concisely

7

Driving force for CISC

- This solution gave rise to another problem:
  - Semantic gap, which is...
    - difference between the operations provided in HLLs and those provided in computer architecture
  - Symptoms of this gap:
    - Execution ineficiency
    - Excessive machine program size
    - Compiler complexity
- Processor designers response:
  - Architectures intended to close this gap, such as...
    - Large instruction sets
    - More addressing modes
    - Hardware implementations of HLL statements
      - e.g. CASE (switch) on VAX

8

Intention of CISC

Complex instruction sets are intended to...

- Ease the task of compiler writer
- Improve execution efficiency
  - Complex operations can be implemented in microcode
- Provide support for more complex HLLs

9

Execution Characteristics

- Studies have been done over the years to determine the characteristics and patterns of execution of machine instructions generated from HLL programs
- The results of these studies inspired some researchers to look for a different approach:
  - To make the architecture that supports the HLL simpler, rather than more complex
- The aspects of computation of interest are as follows:
  - Operations performed
    - Determine the functions to be performed by the processor and its interaction with memory
  - Operands used
    - Determine the memory organization for storing them and the addressing modes for accessing them
  - Execution sequencing
    - Determines the control and pipeline organization

10

Relative Dynamic Frequency of HLL Operations

- A variety of studies have been done to analyze the behavior of HLL programs.
- Dynamic studies are measured during the execution of the program
- The table indicates the relative significance of various statement types in an HLL

	Dynamic Occurrence		Machine-Instruction		Memory-Reference	
	Pascal	C	Pascal	C	Pascal	C
ASSIGN	45%	38%	13%	13%	14%	15%
LOOP	5%	3%	42%	32%	33%	26%
CALL	15%	12%	31%	33%	44%	45%
IF	29%	43%	11%	21%	7%	13%
GOTO	—	3%	—	—	—	—
OTHER	6%	1%	3%	1%	2%	1%

- Assignments
  - Movement of data
- Conditional statements (IF, LOOP)
  - Sequence control
- Procedure call-return is very time consuming
  - Depends on number of parameters passed
  - Depends on level of nesting
- Some HLL instruction lead to many machine code operations

11

Operands

- Table shows dynamic percentage of operands

	Pascal	C	Average
Integer Constant	16%	23%	20%
Scalar Variable	58%	53%	55%
Array/Structure	26%	24%	25%

- Mainly local scalar variables
- Optimisation should concentrate on accessing local variables

12

Implications

- Best support is given by optimising most used and most time consuming features
- Large number of registers
  - Operand referencing
- Careful design of pipelines
  - Branch prediction etc.
- Simplified (reduced) instruction set

Large Register File

- Software solution
  - Require compiler to allocate registers
  - Allocate based on most used variables in a given time
  - Requires sophisticated program analysis
- Hardware solution
  - Have more registers
  - Thus more variables will be in registers

Registers for Local Variables

- Store local scalar variables in registers
- Reduces memory access
- Every procedure (function) call changes locality
- Parameters must be passed
- Results must be returned
- Variables from calling programs must be restored

Global Variables

- There are two options for storing variables declared as global in an HLL:
  - Allocated by the compiler to memory
    - Instructions will use memory-reference operands
    - Inefficient for frequently accessed variables
  - Have a set of registers for global variables
    - Fixed in number
    - Available to all procedures
    - Increased hardware burden
    - Compiler must decide which global variables should be assigned to registers

Registers v Cache

Large Register File	Cache
All local scalars	Recently-used local scalars
Individual variables	Blocks of memory
Compiler-assigned global variables	Recently-used global variables
Save/Restore based on procedure nesting depth	Save/Restore based on cache replacement algorithm
Register addressing	Memory addressing

- Adressing overhead
  - Large based register file is superior

Compiler Based Register Optimization

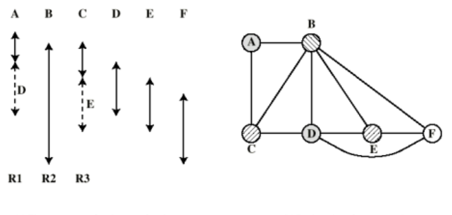
- Assume small number of registers (16-32)
- Optimized register usage is the responsibility of the compiler
- HLL programs have no explicit references to registers
- The objective of compiler is to keep the operands for as many computations as possible in registers to minimize load-and-store operations
- Following approach is taken:
  - Assign symbolic or virtual register to each candidate variable
  - Compiler maps (unlimited) symbolic registers to a fix number of real registers
  - Symbolic registers that do not overlap can share the same real registers
  - If you run out of real registers some variables use memory

Graph Coloring

- The essence of the optimization task is to decide which quantities are to be assigned to registers at any given point in the program
- The technique commonly used in RISC compilers is known as graph coloring:
  - Given a graph of nodes and edges
  - Assign a color to each node such that...
    - Adjacent nodes have different colors
    - Use minimum number of colors
- Graph coloring is adapted to the compiler problem as follows:
  - Nodes of the graph are symbolic registers
  - Two registers that are live in the same program fragment are joined by an edge
  - Try to color the graph with *n* colors, where *n* is the number of real registers
  - Nodes that share the same color can be assigned to the same register
  - Nodes that can not be colored are placed in memory

Graph Coloring Approach

- Assume a program with 6 symbolic registers to be compiled into 3 actual registers
- A possible coloring with 3 colors is indicated.
- One symbolic register, F, is left uncolored and must be used dealt with using loads and stores



(a) Time sequence of active use of registers (b) Register interference graph

Register Optimization

- There is a trade of between...
  - Use of large set of registers and...
  - Compiler-based register optimization
- In one study, reported that...
  - With a simple register optimization, there is little benefit to the use of more than 64 registers
  - With reasonably sofisticated register optimization techniques, there is only marginal performance improvement with more than 32 registers

Why CISC (1)?

- Two principle reasons:
  - Compiler simplification?
    - Disputed…
    - Complex machine instructions harder to exploit
    - Optimization more difficult
  - Smaller programs?
    - Program takes up less memory but…
    - Memory is now cheap
    - May not occupy less bits, just look shorter in symbolic form
      - More instructions require longer op-codes
      - Register references require fewer bits

Why CISC (2)?

- Faster programs?
  - Bias towards use of simpler instructions
  - More complex control unit
  - Microprogram control store larger
  - thus simple instructions take longer to execute

- It is far from clear that CISC is the appropriate solution

RISC Characteristics

- Common characteristics of RISC are:
  - One instruction per cycle
  - Register to register operations
  - Few, simple addressing modes
  - Few, simple instruction formats
  - Hardwired design (no microcode)
  - Fixed instruction format
  - More compile time/effort

RISC v CISC

- Not clear cut
- Many designs borrow from both philosophies
- More recent RISC designs are no longer pure RISC
- More recent CISC designs are no longer pure CISC
  - e.g. PowerPC and Pentium II

RISC Pipelining

- Most instructions are register to register
- An instruction cycle has two phases of execution:
  - I: Instruction fetch
  - E: Execute
    - Performs an ALU operation with register input and output
- For load and store operations, there are three stages:
  - I: Instruction fetch
  - E: Execute
    - Calculates memory address
  - D: Memory
    - Register to memory or memory to register operation

Sequential execution

Load	$rA \leftarrow M$	I	E	D										
Load	$rB \leftarrow M$				I	E	D							
Add	$rC \leftarrow rA + rB$							I	E					
Store	$M \leftarrow rC$									I	E	D		
Branch	X												I	E

- Timing of a sequence of instructions with...
  - No pipelining
- This is a wasteful process

Pipelined timing - I

Load	$rA \leftarrow M$	I	E	D							
Load	$rB \leftarrow M$		I		E	D					
Add	$rC \leftarrow rA + rB$				I		E				
Store	$M \leftarrow rC$						I	E	D		
Branch	X							I		E	
NOOP										I	E

- This is a...
  - Two stage pipelining
  - Provides upto twice the execution rate of a serial scheme
- I and E stages of two different instructions are performed simultaneously
- Two main problems:
  - Only one memory access per stage
    - Assuming that single port main memory is used
  - Branch instruction interrupts the sequencial flow of execution
    - NOOP is inserted by compiler or assembler for minimum circuitry

Pipelined timing - II

Load	$rA \leftarrow M$	I	E	D					
Load	$rB \leftarrow M$		I	E	D				
NOOP				I	E				
Add	$rC \leftarrow rA + rB$				I	E			
Store	$M \leftarrow rC$					I	E	D	
Branch	X						I	E	
NOOP								I	E

- Pipelining can be improved by...
  - permitting two memory access per stage
  - Upto three instructions can be overlapped, prividing...
    - An improvement as much as a factor of 3.
- Branch instruction interrupts the sequencial flow of execution
  - NOOP is inserted by compiler or assembler for minimum circuitry

Pipelined timing - III

Load	$rA \leftarrow M$	I	E_1	E_2	D							
Load	$rB \leftarrow M$		I	E_1	E_2	D						
NOOP				I	E_1	E_2						
NOOP					I	E_1	E_2					
Add	$rC \leftarrow rA + rB$					I	E_1	E_2				
Store	$M \leftarrow rC$						I	E_1	E_2	D		
Branch	X							I	E_1	E_2		
NOOP									I	E_1	E_2	
NOOP										I	E_1	E_2

- Because E stage involves ALU operation, it may be longer.
- Threfore E stage...
  - can be devided into two substages
    - E_1 : register file read
    - E_2 : ALU operation and register write...
- This results in a ...
  - four stage pipeline
  - Maximum speed up of a factor of 4

Optimization of Pipelining

- Pipelining in RISC is efficient.
- The reason for this is that...
  - RISC instructions are simple and regular
- However branch dependencies reduce the overall execution rate
- To compensate for these dependencies ...
  - some code recognition techniques have been developed
- Delayed branch
  - Does not take effect until after execution of following instruction
  - This following instruction is the delay slot
  - Illustrated in next slide

Normal and Delayed Branch

Address	Normal Branch	Delayed Branch	Optimized Delayed Branch
100	LOAD X, rA	LOAD X, rA	LOAD X, rA
101	ADD 1, rA	ADD 1, rA	JUMP 105
102	JUMP 105	JUMP 106	ADD 1, rA
103	ADD rA, rB	NOOP	ADD rA, rB
104	SUB rC, rB	ADD rA, rB	SUB rC, rB
105	STORE rA, Z	SUB rC, rB	STORE rA, Z
106		STORE rA, Z	

Traditional pipeline

Time →

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

- JUMP instruction is fetched at time 4.
- At time 5, JUMP is executed and ADD is fetched
  - However, pipeline must be cleared of instruction 103 because of JUMP instruction
- At time 6, instruction 105 is loaded
- There must be a special circuitry to clear pipeline in branch instructions

RISC pipeline with inserted NOOP

Time →

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I	E					
102 JUMP 106			I	E				
103 NOOP				I	E			
106 STORE rA, Z					I	E	D	

- Because a NOOP inserted, there is no need for special circuitry to clear pipeline
  - NOOP executes with no effect

Reversed instructions

Time →

	1	2	3	4	5	6	7	8
100 LOAD X, Ar	I	E	D					
101 JUMP 105		I	E					
102 ADD 1, rA			I	E				
105 STORE rA, Z				I	E	D		

- JUMP is fetched at time 2, before the ADD instruction
- However, ADD is fetched before JUMP is executed
- At time 4, ADD is executed at the same time that instruction 105 is fetched

Use of Delayed Branch

Time →

	1	2	3	4	5	6	7	8
100 LOAD X, rA	I	E	D					
101 ADD 1, rA		I		E				
102 JUMP 105				I	E			
103 ADD rA, rB					I			
105 STORE rA, Z						I	E	D

(a) Traditional Pipeline

100 LOAD X, rA	I	E	D				
101 ADD 1, rA		I	E				
102 JUMP 106			I	E			
103 NOOP				I	E		
106 STORE rA, Z					I	E	D

(b) RISC Pipeline with Inserted NOOP

100 LOAD X, Ar	I	E	D			
101 JUMP 105		I	E			
102 ADD 1, rA			I	E		
105 STORE rA, Z				I	E	D

(c) Reversed Instructions

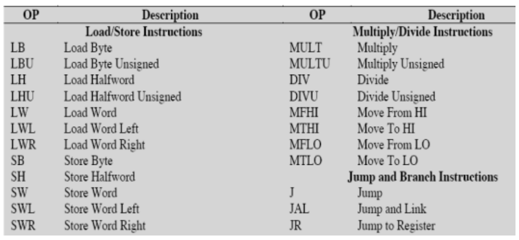
Controversy

- Quantitative
  - compare program sizes and execution speeds
- Qualitative
  - examine issues of high level language support and use of VLSI real estate
- Problems
  - No pair of RISC and CISC that are directly comparable
  - No definitive set of test programs
  - Difficult to separate hardware effects from complier effects
  - Most comparisons done on “toy” rather than production machines
  - Most commercial devices are a mixture

MIPS R4000

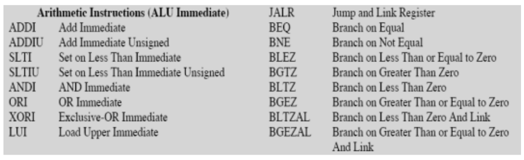
- 1st commercial RISC chip developed by MIPS Technology inch
- MIPS R2000 and R3000, are 32 bit RISC processors
- MIPS R4000 is 64 bit processor
- R4000 is partitioned into two sections:
  - One contains CPU
  - The other contains the coprocessor for memory management
- 32 64 bit registers
- Upto 128 Kbytes of high-speed cache

Instruction set



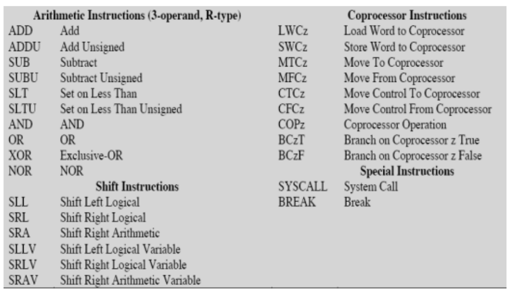
OP	Description	OP	Description
	Load/Store Instructions		**Multiply/Divide Instructions**
LB	Load Byte	MULT	Multiply
LBU	Load Byte Unsigned	MULTU	Multiply Unsigned
LH	Load Halfword	DIV	Divide
LHU	Load Halfword Unsigned	DIVU	Divide Unsigned
LW	Load Word	MFHI	Move From HI
LWL	Load Word Left	MTHI	Move To HI
LWR	Load Word Right	MFLO	Move From LO
SB	Store Byte	MTLO	Move To LO
SH	Store Halfword		**Jump and Branch Instructions**
SW	Store Word	J	Jump
SWL	Store Word Left	JAL	Jump and Link
SWR	Store Word Right	JR	Jump to Register

Instruction set



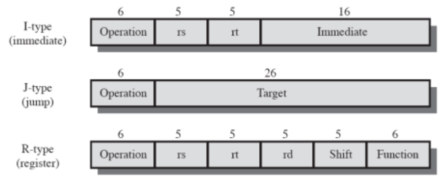
	Arithmetic Instructions (ALU Immediate)	JALR	Jump and Link Register
ADDI	Add Immediate	BEQ	Branch on Equal
ADDIU	Add Immediate Unsigned	BNE	Branch on Not Equal
SLTI	Set on Less Than Immediate	BLEZ	Branch on Less Than or Equal to Zero
SLTIU	Set on Less Than Immediate Unsigned	BGTZ	Branch on Greater Than Zero
ANDI	AND Immediate	BLTZ	Branch on Less Than Zero
ORI	OR Immediate	BGEZ	Branch on Greater Than or Equal to Zero
XORI	Exclusive-OR Immediate	BLTZAL	Branch on Less Than Zero And Link
LUI	Load Upper Immediate	BGEZAL	Branch on Greater Than or Equal to Zero And Link

Instruction set



	Arithmetic Instructions (3-operand, R-type)		**Coprocessor Instructions**
ADD	Add	LWCz	Load Word to Coprocessor
ADDU	Add Unsigned	SWCz	Store Word to Coprocessor
SUB	Subtract	MTCz	Move To Coprocessor
SUBU	Subtract Unsigned	MFCz	Move From Coprocessor
SLT	Set on Less Than	CTCz	Move Control To Coprocessor
SLTU	Set on Less Than Unsigned	CFCz	Move Control From Coprocessor
AND	AND	COPz	Coprocessor Operation
OR	OR	BCzT	Branch on Coprocessor z True
XOR	Exclusive-OR	BCzF	Branch on Coprocessor z False
NOR	NOR		**Special Instructions**
	Shift Instructions	SYSCALL	System Call
SLL	Shift Left Logical	BREAK	Break
SRL	Shift Right Logical		
SRA	Shift Right Arithmetic		
SLLV	Shift Left Logical Variable		
SRLV	Shift Right Logical Variable		
SRAV	Shift Right Arithmetic Variable		

MIPS instruction formats



Operation	Operation code
rs	Source register specifier
rt	Source/destination register specifier
Immediate	Immediate, branch, or address displacement
Target	Jump target address
rd	Destination register specifier
Shift	Shift amount
Function	ALU/shift function specifier