CSCI 4717/5717
Computer Architecture

Topic: RISC Processors

Reading: Stallings, Chapter 13

Major Advances

A number of advances have occurred since the von Neumann architecture was proposed:

- Microprocessors
- Solid-state RAM
- Family concept – separating architecture of machine from implementation

Major Advances (continued)

- Microprogrammed unit
  - Microcode allows for simple programs to be executed from firmware as an action for an instruction
  - Microcode eases the task of designing and implementing the control unit
- Cache memory – speeds up memory hierarchy
- Pipelining – reduces percentage of idle components
- Multiple processors – Speed through parallelism

Semantic Gap

- Difference between operations performed in HLL and those provided by architecture
- Example: Assembly language level case/switch on VAX in hardware
- Problems
  - inefficient execution of code
  - excessive machine program code size
  - increased complexity of compilers
- Predominate operations
  - Movement of data
  - Conditional statements

Measuring Effects of Instructions

- Dynamic occurrence – relative number of times instructions tended to occur in a compiled program
- Static occurrence – counting the number of times they are seen in a program (This is a useless measurement)
- Machine-Instruction Weighted – relative amount of machine code executed as a result of this instruction (based on dynamic occurrence)
- Memory Reference Weighted – relative amount of memory references executed as a result of this instruction (based on dynamic occurrence)
- Procedure call is most time consuming

Operations (continued)

	Dynamic Occurrence		Machine-Instruction Weighted		Memory-Reference Weighted	
	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%

Table 13.2 from Stallings

Operands

- Integer constants
- Scalars (80% of scalars were local to procedure)
- Array/structure
- Lunde, A. "Empirical Evaluation of Some Features of Instruction Set Processor Architectures." Communications of the ACM, March 1977.
  - Each instruction references 0.5 operands in memory
  - Each instruction references 1.4 registers
  - These numbers depend highly on architecture (e.g., number of registers, etc.)

Operands (continued)

Table 13.3 from Stallings

	Pascal	C	Average
Integer constant	16%	23%	20%
Scalar variable	58%	53%	55%
Array/ structure	26%	24%	25%

Procedure calls

Table 13.4 from Stallings

Percentage of Executed Procedure Calls With	Compiler, Interpreter, and Typesetter	Small Nonnumeric Programs
>3 arguments	0–7%	0–5%
>5 arguments	0–3%	0%
>8 words of arguments and local scalars	1–20%	0–6%
>12 words of arguments and local scalars	1–6%	0–3%

This implies that the number of words required when calling a procedure is not that high.

Results of Research

This research suggests:

- Trying to close semantic gap (CISC) is not necessarily answer to optimizing processor design
- A set of general techniques or architectural characteristics can be developed to improve performance.

Reduced Instruction Set Computer (RISC)

Characteristics of a RISC architecture (reduced instruction set is not the only one):

- Limited/simple instruction set – Will become clearer later
- Large number of general-purpose registers and/or use of compiler designed to optimize use of registers – This saves operand referencing
- Optimization of pipeline due to better instruction design – Due to high proportion of conditional branch and procedure call instructions

Increasing Register Availability

There are two basic methods for improving register use

- Software – relies on compiler to maximize register usage
- Hardware – simply create more registers

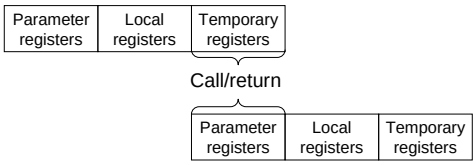
Register Windows

- The hardware solution for making more registers available for a process is to increase the number of registers
  - May slow decoding
  - Should decrease number of memory accesses
- Allocate registers first to local variables
- A procedural call will force registers to be saved into fast memory (cache)
- As shown in Table 13.4 (slide 9), only a small number of parameters and local variables are typically required

Register Windows (continued)

Solution – Create multiple sets of registers, each assigned to a different procedure

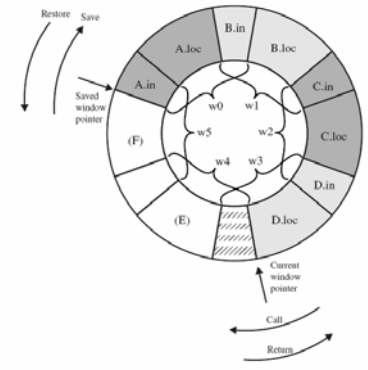
- Saves having to store/retrieve register values from memory
- Allow adjacent procedures to overlap allowing for parameter passing



Register Windows (continued)

- This implies no movement of data to pass parameters.
- Begin to see why compiler writers would make better processor architects
- To make number of registers appear unbounded, architecture should allow for older activations to be stored in memory

Register Windows (continued)



Register Windows (continued)

- When we need to free up a window, an interrupt occurs to store oldest window
- Only need to store parameter registers and local registers
- Temporary registers are associated with parameter registers of next call
- Interrupt is used to restore window after newest function completes
- N-window register file can only hold N-1 procedure activations
- Research showed that N=8 → 1% save or restore of the calls and returns.

Register Windows – Global Variables

- Question: Where do we put global variables?
- Could set global variables in memory
- For often accessed global variables, however, this is inefficient
- Solution: Create an additional set of registers for global variables. (Fixed number and available to all procedures)

Problems with Register Windows

- Increased hardware burden
- Compiler needs to determine which variables get the nice, high-speed registers and which go to memory

Register Windows versus Cache

- It could be said that register windows are similar to a high-speed memory or cache for procedure data
- This is not necessarily a valid comparison

Register Windows versus Cache (continued)

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

Register Windows versus Cache (continued)

There are some areas where caches are more efficient

- They contain data that is definitely used
- Register file may not be fully used by procedure
- Savings in other areas such as code accesses are possible with cache whereas register file only works with local variables

Register Windows versus Cache (continued)

- There are, however, some areas where the register windows are a better choice
  - Register file more closely mimics software which typically operates within a narrow range of procedure calls whereas caches may thrash under certain circumstances
  - Register file wins the speed war when it comes to decoding logic
  - Good compiler design can take better advantage of register window than cache
- Solution – use register file and instructions-only cache

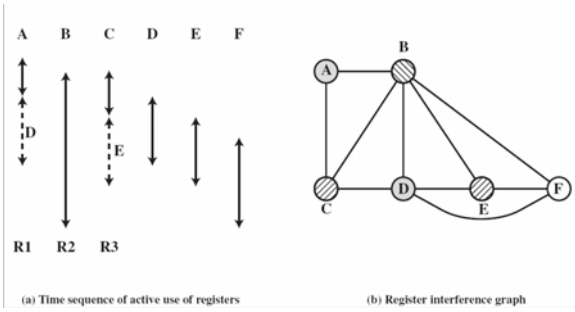
Compiler-based register optimisation

- Assume a reduced number of available registers
- HLL do not use explicit references to registers
- Solution
  - Assign symbolic or virtual register designations to each declared variable
  - Map limited registers to symbolic registers
  - Symbolic registers that do not overlap should share register
  - Load-and-store operations for quantities that overflow number of available registers
- Goal is to decide which quantities are to be assigned registers at any given point in program – "graph coloring"

Graph Coloring

- Technique borrowed from discipline of topology
- Create graph – Register Interference Graph
  - Each vertex is a symbolic register
  - Two symbolic registers that used during the same program fragment are joined by an edge to depict interference
  - Two symbolic vertices linked must have different "colors", i.e., will have to use different registers
  - Goal is to avoid "number of colors" exceeding number of available registers
  - Symbolic registers that go past number of actual registers must be stored in memory

Graph Coloring (continued)



CISC versus RISC

- Complex instructions are possibly more difficult to directly associate w/a HLL instruction – many compilers may just take the simpler, more reliable way out
- Optimization more difficult with complex instructions
- Compilers tend to favor more general, simpler commands, so savings in terms of speed may not be realized either

CISC versus RISC (continued)

CISC programs may take less memory

- Not necessarily an advantage with cheap memory
- Is an advantage due to fewer page faults
- May only be shorter in assembly language view, not necessarily from the point of view of the number of bits in machine code

Additional Design Distinctions

- Further characteristics of RISC
  - One instruction per cycle
  - Register-to-register operations
  - Simple addressing modes
  - Simple instruction formats
- There is no clear-cut design for one or the other
- Many processors contain characteristics of both RISC and CISC

RISC – One Instruction per Cycle

- Cycle = machine cycle
- Fetch two operands from registers – very simple addressing mode
- Perform an ALU operation
- Store the result in a register
- Microcode should not be necessary at all – hardwired code
- Format of instruction is fixed and simple to decode
- Burden is placed on compiler rather than processor – compiler runs once, application runs many times

RISC – Register-to-Register Operations

- Only LOAD and STORE operations should access memory
- ADD Example:
  - RISC – ADD and ADD with carry
  - VAX – 25 different ADD instructions

Simple addressing modes

- Register
- Displacement
- PC-relative
- No indirect addressing – requires two memory accesses
- No more than one memory addressed operand per instruction
- Unaligned addressing not allowed, i.e., addressing only on breaks of 2 or 4
- Simplifies control unit

Simple instruction formats

- Instruction length is fixed – typically 4 bytes
- One or a few formats are used
- Instruction decoding and register operand decoding occurs at the same time
- Simplifies control unit

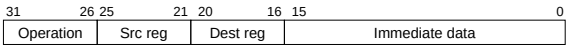
Characteristics of Some Processors

Processor	Number of instruction sizes	Max instruction size in bytes	Number of addressing modes	Indirect addressing	Load/store combined with arithmetic	Max number of memory operands	Unaligned addressing allowed	Max Number of MMU uses	Number of bits for integer register specifier	Number of bits for FP register specifier
AMD29000	1	4	1	no	no	1	no	1	8	3[a]
MIPS R2000	1	4	1	no	no	1	no	1	5	4
SPARC	1	4	2	no	no	1	no	1	5	4
MC88000	1	4	3	no	no	1	no	1	5	4
HP PA	1	4	10[a]	no	no	1	no	1	5	4
IBM RT/PC	2[a]	4	1	no	no	1	no	1	4[a]	3[a]
IBM RS/6000	1	4	4	no	no	1	yes	1	5	5
Intel i860	1	4	4	no	no	1	no	1	5	4
IBM 3090	4	8	2[b]	no[a]	yes	2	yes	4	4	2
Intel 80486	12	12	15	no[b]	yes	2	yes	4	3	3
NSC 32016	21	21	23	yes	yes	2	yes	4	3	3
MC68040	11	22	44	yes	yes	2	yes	8	4	3
VAX	56	56	22	yes	yes	6	yes	24	4	0
Clipper	4[a]	8[a]	9[a]	no	no	1	0	2	4[a]	3[a]
Intel 80960	2[a]	8[a]	9[a]	no	no	1	yes[a]	—	5	3[a]

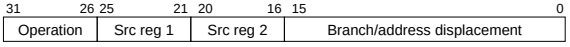
a RISC that does not conform to this characteristic.
b CISC that does not conform to this characteristic.

MIPS Instruction Format (Fig. 13.8)

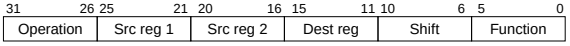
Register-Immediate Instruction



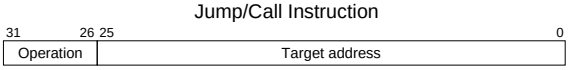
Branch Instruction



Register-to-Register Instruction



Jump/Call Instruction



MIPS Instruction Format (continued)

- What is the largest immediate integer that can be subtracted from a register?
- How far away from the current instruction can a branch instruction go?
- What is the memory range for a jump or call instruction?
- Why might a branch operation require two registers instead of referencing flags?

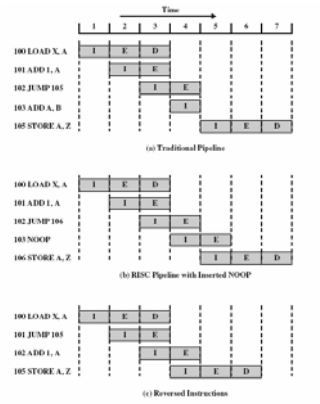
Delayed Branch

- Traditional pipelining disposes of instruction loaded in pipe after branch
- Delayed branching executes instruction loaded in pipe after branch
- NOOP can be used if instruction cannot be found to execute after JUMP. This makes it so no special circuitry is needed to clear the pipe.
- It is left up to the compiler to rearrange instructions or add NOOPs

Delayed Branch (continued)

Address	Normal Branch	Delayed Branch	Optimized Delayed Branch
100	LOAD X,A	LOAD X,A	LOAD X,A
101	ADD 1,A	ADD 1,A	JUMP 105
102	JUMP 105	JUMP 106	ADD 1,A
103	ADD A,B	NOOP	ADD A,B
104	SUB C,B	ADD A,B	SUB C,B
105	STORE A,Z	SUB C,B	STORE A,Z
106		STORE A,Z	

Delayed Branch (continued)



Delayed Load

- Similar to delayed branch in that an instruction that doesn't use register being loaded can execute during the D phase of a load instruction
- During a load, processor "locks" register being loaded and continues execution until instruction requiring locked register is referenced
- Left up to the compiler to rearrange instructions

Problem 13.6 from Textbook

```
    S := 0;
    for K :=1 to 100 do  S := S – K;

                -- translates to --

        LD    R1, 0          ;keep value of S in R1
        LD    R2, 1          ;keep value of K in R2
LP      SUB   R1, R1, R2     ;S := S – K
        BEQ   R2, 100, EXIT  ;done if K = 100
        ADD   R2, R2, 1      ;else increment K
        JMP   LP             ;back to start of loop
```

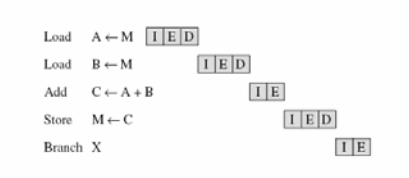
Where should the compiler add NOOPs or rearrange instructions?

RISC Pipelining

- Pipelining structure is simplified greatly thus making delay between stages much less apparent and simplifying logic of the stages
- ALU operations
  - I: instruction fetch
  - E: execute (register-to-register)
- Load and store operations
  - I: instruction fetch
  - E: execute (calculates memory address)
  - D: Memory (register-to-memory or memory-to-register operations)

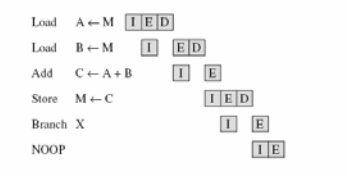
Comparing the Effects of Pipelining

Sequential execution – obviously inefficient



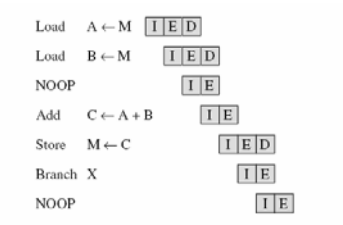
Comparing the Effects of Pipelining (continued)

- Two-way pipelined timing – I and E stages of two different instructions can be performed simultaneously
- Yields up to twice the execution rate of sequential
- Problems
  - Causes wait state with accesses to memory
  - Branch disrupts flow (NOOP instruction can be inserted by assembler or compiler)



Comparing the Effects of Pipelining (continued)

Permitting two memory accesses at one time allows for fully pipelined operation (dual-port RAM)



Comparing the Effects of Pipelining (continued)

- Since E is usually longer, break E into two parts
  - E1 – register file read
  - E2 – ALU operation and register write
- Because of RISC design, this is not as difficult to do and up to four instructions can be under way at one time (potential speedup of 4)