REDUCED INSTRUCTION SET COMPUTERS (RISC)

1. Why do we need RISCs?
2. Some Typical Features of Current Programs
3. Main Characteristics of RISC Architectures
4. Are RISCs Really Better than CISCs?
5. Examples

What are RISCs and why do we need them?

- RISC architectures represent an important innovation in the area of computer organization.
- The RISC architecture is an attempt to produce more CPU power by simplifying the instruction set of the CPU.
- The opposed trend to RISC is that of complex instruction set computers (CISC).

Both RISC and CISC architectures have been developed as an attempt to cover the semantic gap.

The Semantic Gap

Problem: How should new HLL programs be compiled and executed efficiently on a processor architecture?

Two possible answers:

1. **The CISC approach**: design very complex architectures including a large number of instructions and addressing modes; include also instructions close to those present in HLL.
2. **The RISC approach**: simplify the instruction set and adapt it to the real requirements of user programs.

In order to improve the efficiency of software development, new and powerful programming languages have been developed (Ada, C++, Java). They provide: high level of abstraction, conciseness, power.

By this evolution the semantic gap grows.
**Evaluation of Program Execution**  
**or**  
**What are Programs Doing Most of the Time?**

- Several studies have been conducted to determine the execution characteristics of machine instruction sequences generated from HLL programs.

- Aspects of interest:
  1. The frequency of operations performed.
  2. The types of operands and their frequency of use.
  3. Execution sequencing (frequency of jumps, loops, subprogram calls).

---

**Frequency of Instructions Executed**

- **Frequency distribution of executed machine instructions:***
  - moves: 33%
  - conditional branch: 20%
  - Arithmetic/logic: 16%
  - others: Between 0.1% and 10%

- **Addressing modes:** the overwhelming majority of instructions uses simple addressing modes, in which the address can be calculated in a single cycle (register, register indirect, displacement); complex addressing modes (memory indirect, indexed+indirect, displacement+indexed, stack) are used only by ~18% of the instructions.

---

**Operand Types**

- 74 to 80% of the operands are **scalars** (integers, reals, characters, etc.) which can be hold in registers;
- the rest (20-26%) are arrays/structures; 90% of them are **global** variables;
- 80% of the scalars are **local** variables.

The majority of operands are local variables of scalar type, which can be stored in registers.

---

**Procedure Calls**

Investigations have been also performed about the percentage of the total execution time spent executing a certain HLL instruction.

- It turned out that comparing HLL instructions, most of the time is spent executing CALLs and RETURNs.
- Even if only 15% of the executed HLL instructions is a CALL or RETURN, they are executed most of the time, because of their complexity.
- A CALL or RETURN is compiled into a relatively long sequence of machine instructions with a lot of memory references.

---

**Procedure Calls (cont’d)**

```c
int x, y; float z;

void proc3(int a, int b, int c) {
    // ...}
}

void proc2(int k) {
    int j, q;
    proc3(j, 5, q);
}

void proc1() {
    int i;
    proc2(i);
}

main () {
    proc1();
}
```

Some statistics concerning procedure calls:

- Only 1.25% of called procedures have more than six parameters.
- Only 6.7% of called procedures have more than six local variables.
- Chains of nested procedure calls are usually short and only very seldom longer than 6.
Conclusions from Evaluation of Program Execution

- An overwhelming preponderance of simple (ALU and move) operations over complex operations.
- Preponderance of simple addressing modes.
- Large frequency of operand accesses; on average each instruction references 1.9 operands.
- Most of the referenced operands are scalars (so they can be stored in a register) and are local variables or parameters.
- Optimizing the procedure CALL/RETURN mechanism promises large benefits in speed.

These conclusions have been at the starting point to the Reduced Instruction Set Computer (RISC) approach.

Main Characteristics of RISC Architectures

- The instruction set is limited and includes only simple instructions.
  - The goal is to create an instruction set containing instructions that execute quickly; most of the RISC instructions are executed in a single machine cycle (after fetched and decoded).
  - Pipeline operation (without memory reference):

Main Characteristics of RISC Architectures (cont’d)

Let’s see some small example:

Assume:
- we have a program with 80% of executed instructions being simple and 20% complex;
- on a CISC machine simple instructions take 4 cycles, complex instructions take 8 cycles; cycle time is 100 ns ($10^{-7}$ s);
- on a RISC machine simple instructions are executed in one cycle; complex operations are implemented as a sequence of instructions; we consider on average 14 instructions (14 cycles) for a complex operation; cycle time is 75 ns ($0.75 \times 10^{-7}$ s).

How much time takes a program of 1 000 000 instructions?

$$\begin{align*}
\text{CISC: } & (10^6 \times 0.80 \times 4 + 10^6 \times 0.20 \times 8) \times 10^{-7} = 0.48 \text{ s} \\
\text{RISC: } & (10^6 \times 0.80 \times 1 + 10^6 \times 0.20 \times 14) \times 0.75 \times 10^{-7} = 0.27 \text{ s}
\end{align*}$$

- complex operations take more time on the RISC, but their number is small;
- because of its simplicity, the RISC works at a smaller cycle time; with the CISC, simple instructions are slowed down because of the increased data path length and the increased control complexity.

Main Characteristics of RISC Architectures (cont’d)

- Load-and-store architecture
  - Only LOAD and STORE instructions reference data in memory; all other instructions operate only with registers (are register-to-register instructions); thus, only the few instructions accessing memory need more than one cycle to execute (after fetched and decoded).
  - Pipeline operation with memory reference:

Instructions use only few addressing modes
  - Addressing modes are usually register, direct, register indirect, displacement.

Instructions are of fixed length and uniform format
  - This makes the loading and decoding of instructions simple and fast; it is not needed to wait until the length of an instruction is known in order to start decoding the following one;
  - Decoding is simplified because opcode and address fields are located in the same position for all instructions.
Main Characteristics of RISC Architectures (cont'd)

- A large number of registers is available
  - Variables and intermediate results can be stored in registers and do not require repeated loads and stores from/to memory.
  - All local variables of procedures and the passed parameters can be stored in registers (see slide 8 for comments on possible number of variables and parameters).

What happens when a new procedure is called?

- Normally the registers have to be saved in memory (they contain values of variables and parameters for the calling procedure); at return to the calling procedure, the values have to be again loaded from memory. This takes a lot of time.
- If a large number of registers is available, a new set of registers can be allocated to the called procedure and the register set assigned to the calling one remains untouched.

Main Characteristics of RISC Architectures (cont’d)

Is the above strategy realistic?

- The strategy is realistic, because the number of local variables in procedures is not large. The chains of nested procedure calls is only exceptionally larger than 6.
- If the chain of nested procedure calls becomes large, at a certain call there will be no registers to be assigned to the called procedure; in this case local variables and parameters have to be stored in memory.

Why is a large number of registers typical for RISC architectures?

- Because of the reduced complexity of the processor there is enough space on the chip to be allocated to a large number of registers. This, usually, is not the case with CISCs.

Main Characteristics of RISC Architectures (cont’d)

The delayed load problem

- LOAD instructions (similar to the STORE) require memory access and their execution cannot be completed in a single clock cycle. However, in the next cycle a new instruction is started by the processor.

  LOAD R1,X
  ADD R2,R1
  ADD R4,R3

Two possible solutions:
1. The hardware should delay the execution of the instruction following the LOAD, if this instruction needs the loaded value
2. A more efficient, compiler based, solution, which has similarities with the delayed branching, is the delayed load:

  LOAD R1,X
  ADD R2,R1
  ADD R4,R3
  SUB R2,R4

The delayed load problem (cont’d)

- With delayed load the processor always executes the instruction following a LOAD, without a stall; It is the programmers (compilers) responsibility that this instruction does not need the loaded value.

The following sequence is not correct with such a processor, and a compiler will not generate something like this:

  LOAD R1,X
  ADD R2,R1
  ADD R4,R3
  SUB R2,R4

The value in R1 is not yet available!
The delayed load problem (cont'd)

The following one is the correct sequence:

```plaintext
LOAD R1,X  loads from address X into R1
ADD R4,R3  R4 ← R4 + R3
ADD R2,R1  R2 ← R2 + R1
SUB R2,R4  R2 ← R2 - R4
```

For the following sequence the compiler has generated a NOP after the LOAD because there is no instruction to fill the load-delay slot:

```plaintext
LOAD R1,X  loads from address X into R1
NOP
ADD R2,R1  R2 ← R2 + R1
ADD R4,R2  R4 ← R4 + R2
STORE R4,X stores from R4 to address X
```

Are RISCs Really Better than CISCs?

- RISC architectures have several advantages and they were discussed throughout this lecture. However, a definitive answer to the above question is difficult to give.
- A lot of performance comparisons have shown that benchmark programs are really running faster on RISC processors than on processors with CISC characteristics.
- However, it is difficult to identify which feature of a processor produces the higher performance. Some "CISC fans" argue that the higher speed is not produced by the typical RISC features but because of technology, better compilers, etc.
- An argument in favour of the CISC: the simpler instruction set of RISC processors results in a larger memory requirement compared to the similar program compiled for a CISC architecture.

Most of the current processors are not typical RISCs or CISCs but try to combine advantages of both approaches.

Some Processor Examples

RISC Architectures:

- **Sun SPARC**
  - Nr. of instructions: 52
  - Instruction size: 4
  - Instruction format: fixed
  - Addressing modes: 2
  - Number of general purpose registers: up to 520

- **PowerPC**
  - Nr. of instructions: 206
  - Instruction size: 4
  - Instruction format: not fixed (but small differences)
  - Addressing modes: 2
  - Number of general purpose registers: 32

ARM

- Nr. of instructions (in the standard set, not the Thumb): 122 (many versions only provide a subset)
- Instruction size: 4 (standard), 2 (Thumb)
- Instruction format: fixed (different for regular and Thumb)
- Addressing modes: 3
- Number of general purpose registers: 27 (16 can be used at any one time)

Current ARM processors can execute both the standard 32-bit instruction set and the 16-bit Thumb instruction set. Thumb contains a subset of the 32-bit set, encoded into 16-bit instructions.

CISC Architectures:

- **VAX 11/780**
  - Nr. of instructions: 303
  - Instruction size: 2 - 57
  - Instruction format: not fixed
  - Addressing modes: 22
  - Number of general purpose registers: 16

- **Pentium**
  - Nr. of instructions: 235
  - Instruction size: 1 - 11
  - Instruction format: not fixed
  - Addressing modes: 11
  - Number of general purpose registers: 8 (32-bit mode), 16 (64-bit mode)
**Summary**

- Both RISCs and CISCs try to solve the same problem: to cover the semantic gap. They do it in different ways. CISCs are going the traditional way of implementing more and more complex instructions. RISCs try to simplify the instruction set.
- Innovations in RISC architectures are based on a close analysis of a large set of widely used programs.
- The main features of RISC architectures are: reduced number of simple instructions, few addressing modes, load-store architecture, instructions are of fixed length and format, a large number of registers is available.
- One of the main concerns of RISC designers was to maximise the efficiency of pipelining.
- Present architectures often include both RISC and CISC features.