CMSC 611: Advanced Computer Architecture

Benchmarks & Instruction Set Architecture

Some material adapted from Mohamed Younis, UMBC CMSC 611 Spr 2003 course slides

Using MIPS

- MIPS = Million of Instructions Per Second
  - one of the simplest metrics
  - valid only in a limited context

$$\text{MIPS (native MIPS)} = \frac{\text{Instruction count}}{\text{Execution time} \times 10^6}$$

- There are three problems with MIPS:
  - MIPS specifies the instruction execution rate but not the capabilities of the instructions
  - MIPS varies between programs on the same computer
  - MIPS can vary inversely with performance (see next example)

The use of MIPS is simple and intuitive, faster machines have bigger MIPS

Example

Consider the machine with the following three instruction classes and CPI:

Instruction class	CPI for this instruction class
A	1
B	2
C	3

Now suppose we measure the code for the same program from two different compilers and obtain the following data:

Code from	Instruction count in (billions) for each instruction class		
	A	B	C
Compiler 1	5	1	1
Compiler 2	10	1	1

Assume that the machine's clock rate is 500 MHz. Which code sequence will execute faster according to MIPS? According to execution time?

Answer:

Using the formula: $$\text{CPU clock cycles} = \sum_{i=1}^{n} CPI_i \times C_i$$

Sequence 1: CPU clock cycles = $(5 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 10 \times 10^9$ cycles

Sequence 2: CPU clock cycles = $(10 \times 1 + 1 \times 2 + 1 \times 3) \times 10^9 = 15 \times 10^9$ cycles

Example (Cont.)

Using the formula: $\text{Exection time} = \dfrac{\text{CPU clock cycles}}{\text{Clock rate}}$

Sequence 1: Execution time = $(10\times10^9)/(500\times10^6)$ = 20 seconds
Sequence 2: Execution time = $(15\times10^9)/(500\times10^6)$ = 30 seconds

Therefore compiler 1 generates a faster program

Using the formula: $\text{MIPS} = \dfrac{\text{Instruction count}}{\text{Execution time} \times 10^6}$

Sequence 1: $\text{MIPS} = \dfrac{(5+1+1)\times10^9}{20\times10^6} = 350$

Sequence 2: $\text{MIPS} = \dfrac{(10+1+1)\times10^9}{30\times10^6} = 400$

Although compiler 2 has a higher MIPS rating, the code from generated by compiler 1 runs faster

Native, Peak and Relative MIPS, & FLOPS

- Peak MIPS is obtained by choosing an instruction mix that maximizes the CPI, even if the mix is impractical
- To make MIPS more practical among different instruction sets, a relative MIPS is introduced to compare machines to an agreed-upon reference machine (e.g. Vax 11/780)

$$\text{Relative MIPS} = \frac{\text{Execution time}_{\text{reference}}}{\text{Execution time}_{\text{unrated}}} \times \text{MIPS}_{\text{reference}}$$

Native, Peak and Relative MIPS, & FLOPS

- With the fast development in the computer technology, reference machine cannot be guaranteed to exist
- Relative MIPS is practical for evolving design of the same computer
- With the introduction of supercomputers around speeding up floating point computation, the term MFLOP is introduced analogous to MIPS

Synthetic Benchmarks

- Synthetic benchmarks are artificial programs that are constructed to match the characteristics of large set of programs
- Whetstone (scientific programs in Algol → Fortran) and Dhrystone (systems programs in Ada → C) are the most popular synthetic benchmarks
- Whetstone performance is measured in "Whetstone per second" – the number of executions of one iteration of the whetstone benchmark

Synthetic Benchmark Drawbacks

1. They do not reflect the user interest since they are not real applications
2. They do not reflect real program behavior (e.g. memory access pattern)
3. Compiler and hardware can inflate the performance of these programs far beyond what the same optimization can achieve for real-programs

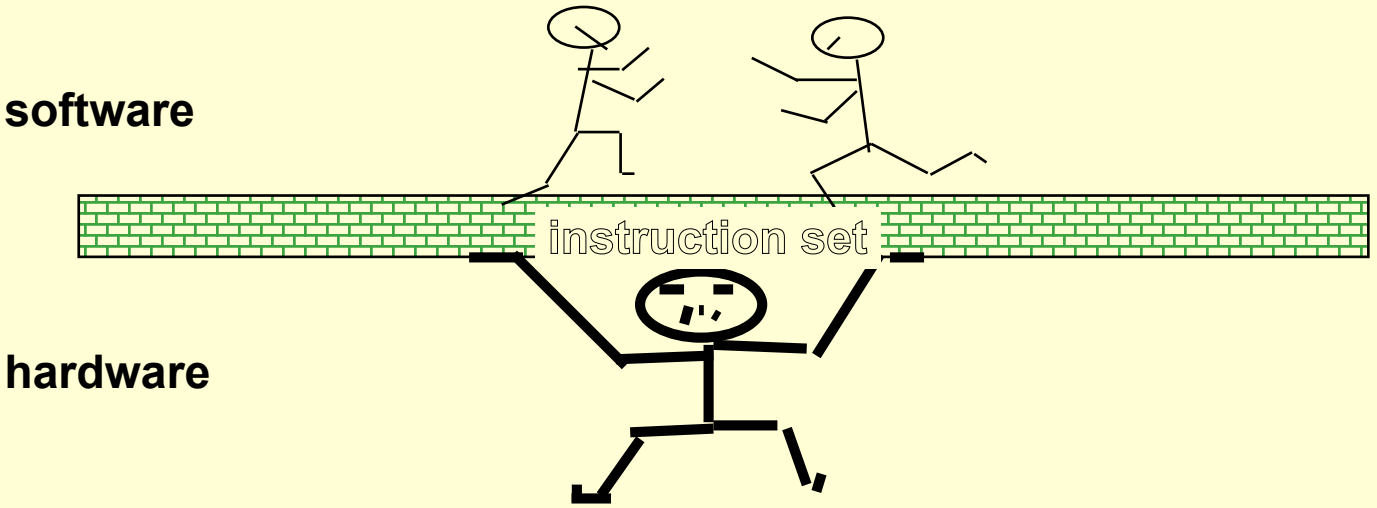
Dhrystone Examples

- By assuming word alignment in string copy a 20-30% performance improvement could be achieved
  - Although 99.70-99.98% of typical string copies could NOT use such optimization
- Compiler optimization could easily discard 25% of the Dhrystone code for single iteration loops and inline procedure expansion

Final Performance Remarks

- Designing for performance only without considering cost is unrealistic
  - In the supercomputing industry performance is the primary and dominant goal
  - Low-end personal and embedded computers are extremely cost driven
- Performance depends on three major factors
  - number of instructions,
  - cycles consumed by instruction execution
  - clock cycle
- The art of computer design lies not in plugging numbers in a performance equation, but in accurately determining how design alternatives will affect performance and cost

Introduction



- To command a computer's hardware, you must speak its language
- Instructions: the “words” of a machine's language
- Instruction set: its “vocabulary
- The MIPS instruction set is used as a case study

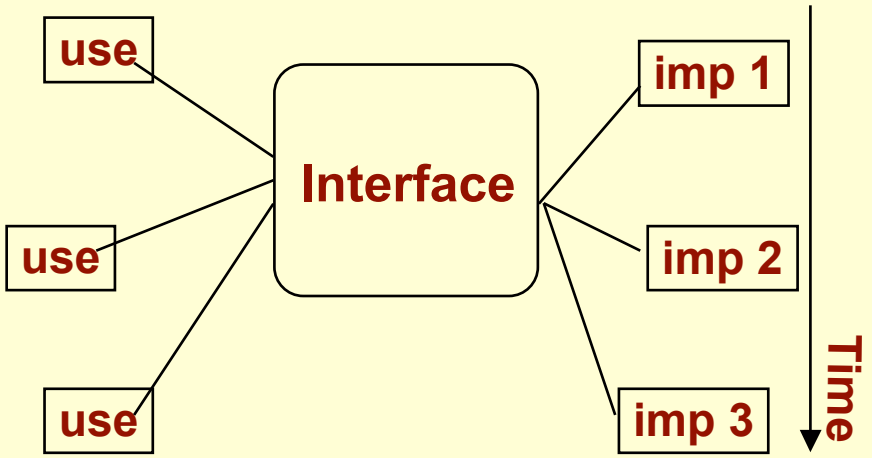
Figure: Dave Patterson

Instruction Set Architecture

- Once you learn one machine language, it is easy to pick up others:
  - Common fundamental operations
  - All designer have the same goals: simplify building hardware, maximize performance, minimize cost
- Goals:
  - Introduce design alternatives
  - Present a taxonomy of ISA alternatives
    - + some qualitative assessment of pros and cons
  - Present and analyze some instruction set measurements
  - Address the issue of languages and compilers and their bearing on instruction set architecture
  - Show some example ISA's

Interface Design

- A good interface:
  - Lasts through many implementations (portability, compatibility)
  - Is used in many different ways (generality)
  - Provides convenient functionality to higher levels
  - Permits an efficient implementation at lower levels
- Design decisions must take into account:
  - Technology
  - Machine organization
  - Programming languages
  - Compiler technology
  - Operating systems



Slide: Dave Patterson

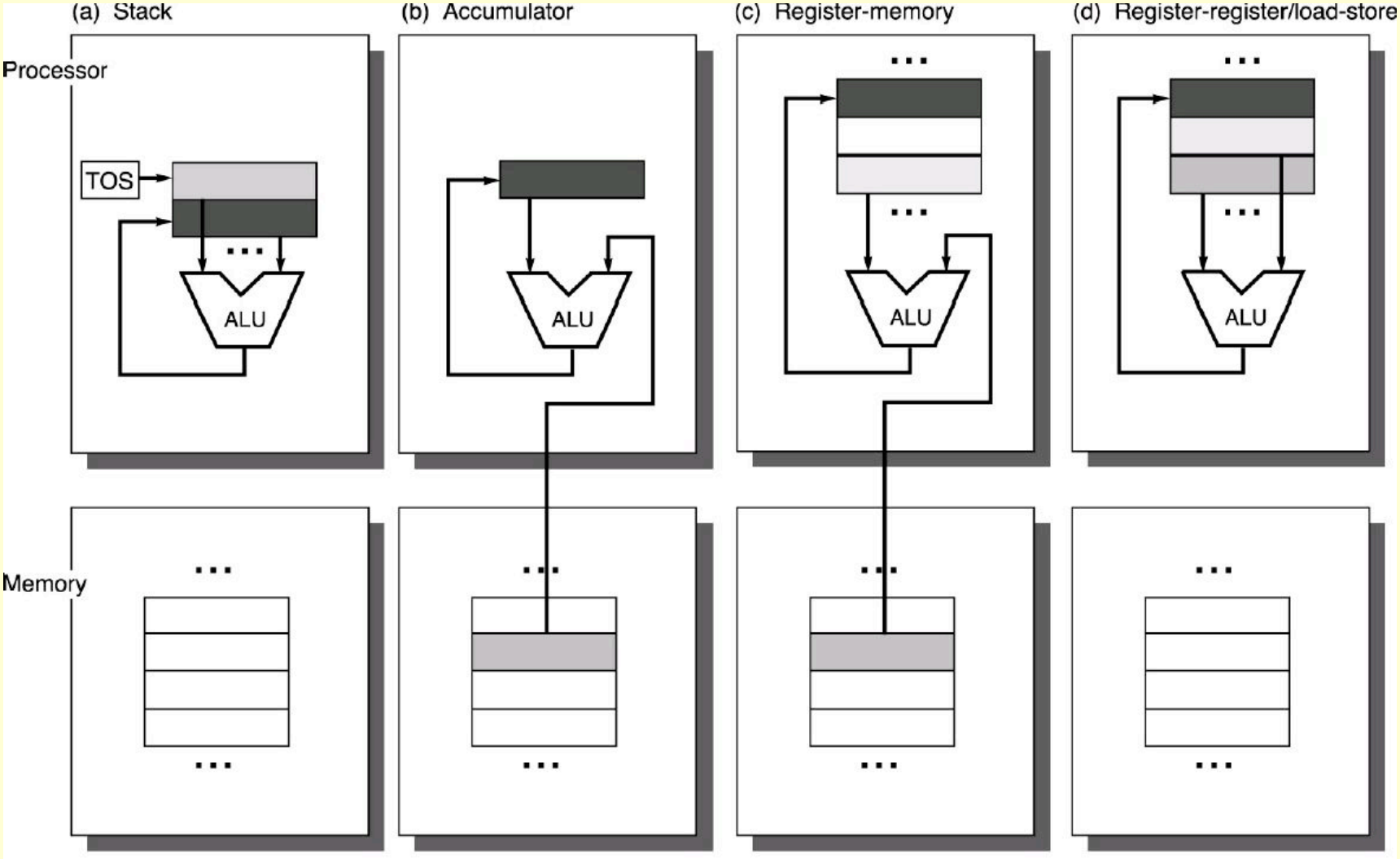
Memory ISAs

- Terms
  - Result = Operand <operation> Operand
- Stack
  - Operate on top stack elements, push result back on stack
- Memory-Memory
  - Operands (and possibly also result) in memory

Register ISAs

- Accumulator Architecture
  - Common in early stored-program computers when hardware was expensive
  - Machine has only one register (accumulator) involved in all math & logic operations
  - Accumulator = Accumulator op Memory
- Extended Accumulator Architecture (8086)
  - Dedicated registers for specific operations, e.g stack and array index registers, added
- General-Purpose Register Architecture (MIPS)
  - Register flexibility
  - Can further divide these into:
    - Register-memory: allows for one operand to be in memory
    - Register-register (load-store): all operands in registers

ISA Operations



Famous ISA

- Stack
- Memory-Memory
- Accumulator Architecture
- Extended Accumulator Architecture
- General-Purpose Register Architecture

Machine	# general-purpose registers	Architecture style	Year
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	32	Register-memory	1985
PowerPC	32	Load-store	1992
DEC Alpha	32	Load-store	1992

Other types of Architecture

- High-Level-Language Architecture
  - In the 1960s, systems software was rarely written in high-level languages
    - virtually every commercial operating system before Unix was written in assembly
  - Some people blamed the code density on the instruction set rather than the programming language
  - A machine design philosophy advocated making the hardware more like high-level languages
  - The effectiveness of high-level languages, memory size limitation and lack of efficient compilers doomed this philosophy to a historical footnote

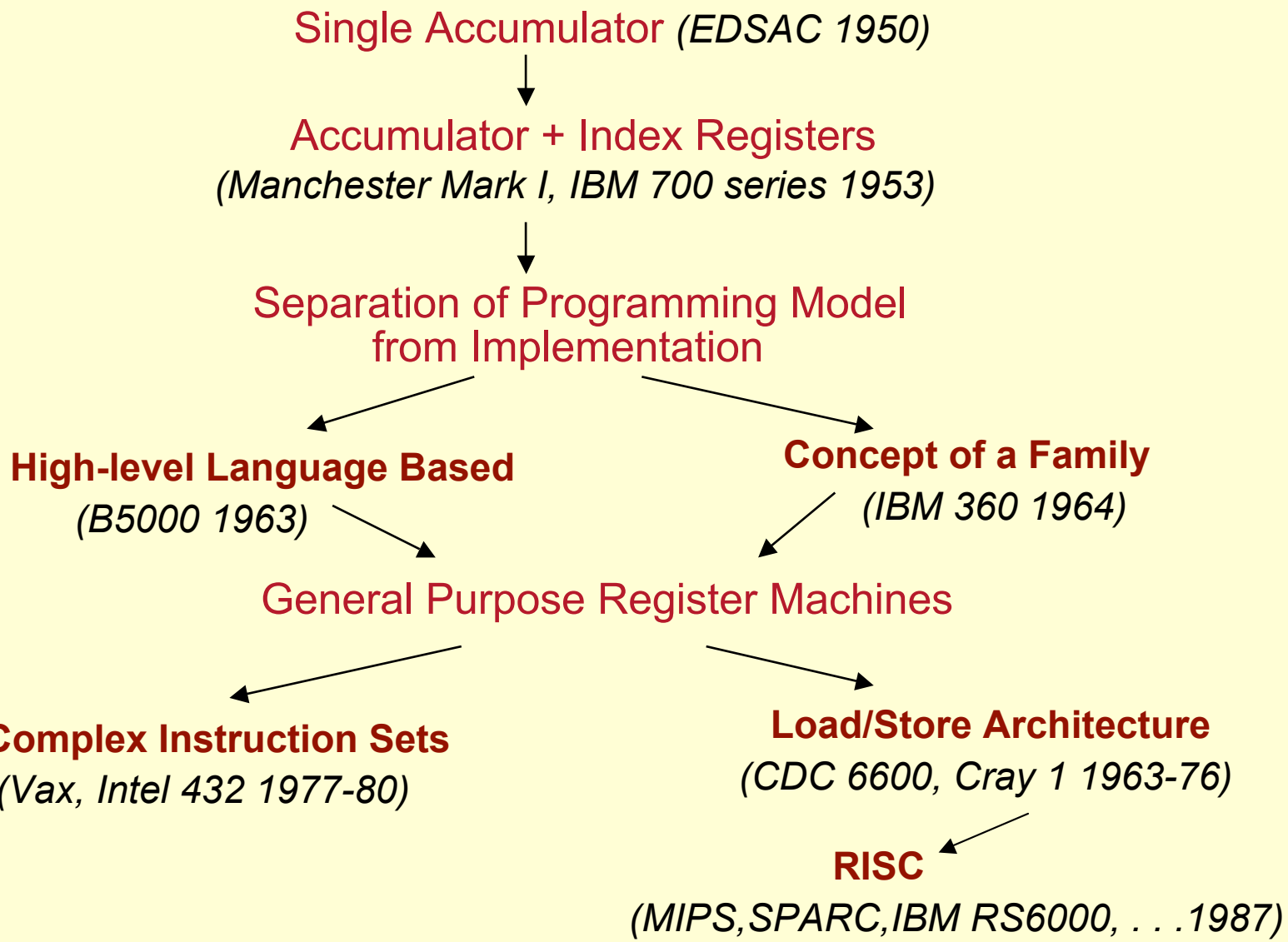
Other types of Architecture

- Reduced Instruction Set Architecture
  - With the recent development in compiler technology and expanded memory sizes less programmers are using assembly level coding
  - Drives ISA to favor benefit for compilers over ease of manual programming
- RISC architecture favors simplified hardware design over rich instruction set
  - Rely on compilers to perform complex operations
- Virtually all new architecture since 1982 follows the RISC philosophy:
  - fixed instruction lengths, load-store operations, and limited addressing mode

Compact Code

- Scarce memory or limited transmit time (JVM)
- Variable-length instructions (Intel 80x86)
  - Match instruction length ot operand specification
  - Minimize code size
- Stack machines abandon registers altogether
  - Stack machines simplify compilers
  - Lend themselves to a compact instruction encoding
  - BUT limit compiler optimization

Evolution of Instruction Sets



Slide: Dave Patterson