CPT_S 260 Intro to Computer Architecture
Lecture 12

Intro to MIPS
February 7, 2022

Ganapati Bhat
School of Electrical Engineering and Computer Science
Washington State University

Instruction Set Architecture

- **Instruction Set Architecture (ISA) serves as the boundary between Hardware and Software**
  - Provides the agreement between programmers and hardware internals
  - *Visible state* of the system
  - Defines how each instruction changes the visible state of the system
- **ISA is used by programmers to model how the hardware works**
  - Emulation before deploying the programs
  - Simulation, such has gem5
- **ISA defines the instructions available and their encodings**
  - Add, subtract, multiply
  - Branch, jump

Architecture v/s Implementation

- **Architecture defines the "what"**
  - The visible state to the programmer
  - The function of each instruction
  - The input and output interface of each instruction
- **Implementation describes the "how"**
  - The number of cycles needed to execute
  - The sequence of steps in the execution
- **What the advantages of separating architecture and implementation?**
  - Compatibility (VAX, ARM)
  - Longevity (x86)
  - Amortize research investment
  - Retain software investment

Instruction Set

- **The repertoire of instructions of a computer**
- **Different computers have different instruction sets**
  - But with many aspects in common
- **You might think that the languages of computers would be as diverse as those of people, but in reality, computer languages are quite similar, more like regional dialects than like independent languages.**
- **The chosen instruction set comes from MIPS Technologies, and is an elegant example of the instruction sets designed since the 1980s.**

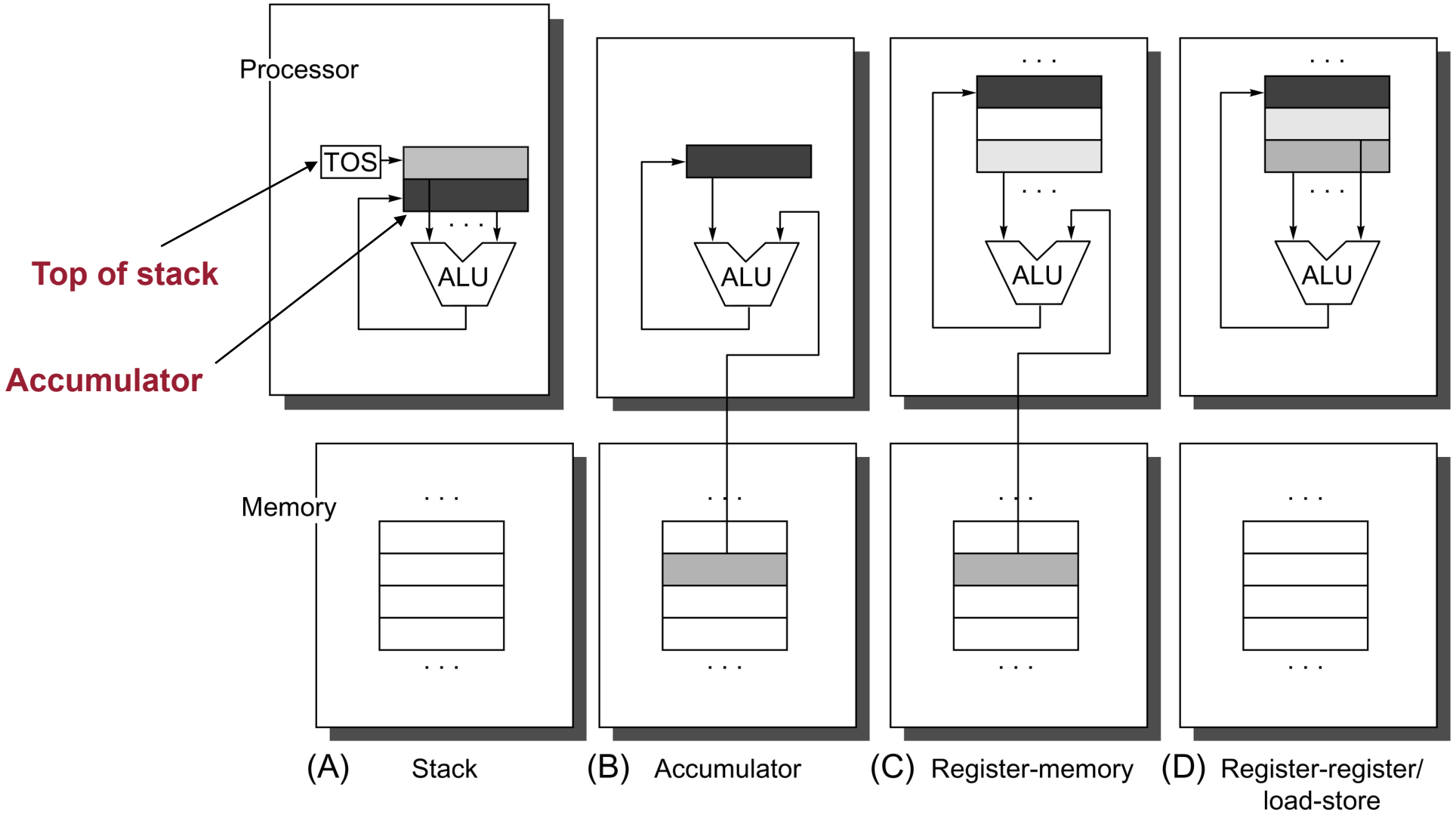
Instruction Set

- ARMv7 is similar to MIPS. More than 9 billion chips with ARM processors were manufactured in 2011
- Intel x86, which powers both the PC and the cloud of the PostPC Era.
- ARMv8, which extends the address size of the ARMv7 from 32 bits to 64 bits. Ironically, as we shall see, this 2013 instruction set is closer to MIPS than it is to ARMv7

ISA Classification

- **Type of *internal storage* is the most basic differentiation**
- **Stack**
  - Operands are implicitly at the top of the stack
- **Accumulator**
  - One operand is the accumulator
- **Register-memory**
  - Operands are from registers and memory
- **Register-register/load-store**
  - Operands are stored in registers
- **Register-memory and register-register also called general-purpose**
  - Explicit operands

ISA Classification Visualization



Code Sequence for C = A + B

- Assume A, B, C belong in memory
- Values of A, B cannot be destroyed

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

- Implicit operands for stack and accumulator
- Explicit operands for register architectures

Load-store Architectures

- **Virtually every new architecture designed since 1980s**
  - Early computers used stack or accumulator architectures
- **Major reasons for rise of general-purpose architectures**
  - Registers are faster than memory
  - Registers are easier to use for compilers
  - E.g. (A*B) – (B– C) – (A * D) can be done in any order
    - » Stack architectures must process left to right
- **Advantages of registers**
  - Process in any order
  - Hold variables
  - Reduces memory traffic
  - Improves code density

MIPs Instruction Goals

- **Showing how it is represented in hardware**
- **The relationship between high-level programming languages and this more primitive one**
  - Examples in C and JAVA
- **See the impact of programming languages and compiler optimization on performance**
- **Writing programs in the language of the computer and running them on the simulator**

Comparison of General-Purpose Architectures

- **Register-register**
  - Advantages
    - » **Simple**, fixed-length instruction encoding
    - » Instructions take similar number of clock cycles to execute
  - Disadvantages
    - » Higher instruction count when compared to ISAs with memory references
    - » More instructions and lower instruction density leads to larger programs
- **Register-memory**
  - Advantages
    - » Data access **without separate load** instruction
    - » Instructions are **easy to encode** and yields good code density
  - Disadvantages
    - » Operands are not equivalent since source operand in a binary operation is destroyed
    - » Clocks per instructions vary by operand location
    - » Encoding register and a memory address in each instruction may restrict number of registers

Comparison of General-Purpose Architectures (2)

- **Memory-memory**
  - Advantages
    - » Most compact
    - » Doesn't waste registers for temporary values
  - Disadvantages
    - » Large variation in instruction size, especially for three-operand instructions
    - » Large variation in work per instruction
    - » Memory access create memory bottleneck
- **Memory-memory architectures not used today**

Issues in Instruction Set Design

- Memory addressing
- Type and size of operands
- Operations in the instruction set
- Encoding of ISAs
- Implementation (pipelining, scheduling, etc.)

Assembly Language

- **An understandable representation of different micro level instruction**
  - Easily convertible into machine language
- **Closer to the hardware compare to high level programming languages(e.g., c and c++)**
- **Can be used beside high level languages to enhance the performance!**

Stored-program concept

- The idea that instructions and data of many types can be stored in memory as numbers, leading to the stored- program computer.

- Created by Jon von Neumann in the late 1940s.

Arithmetic Operations

- **Add and subtract, three operands**
  - Two sources and one destination

```
add a, b, c  # a gets b + c
```

- **All arithmetic operations have this form**
- ***Design Principle 1:* Simplicity favors regularity**
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost

Arithmetic Example

- C code:

```
f = (g + h) - (i + j);
```

- Compiled MIPS code:

```
add t0, g, h   # temp t0 = g + h
add t1, i, j   # temp t1 = i + j
sub f, t0, t1  # f = t0 - t1
```

- We see what are these variables!

Register Operands

- **Arithmetic instructions use register operands**
- **MIPS has a 32 × 32-bit register file**
  - Use for frequently accessed data
  - Numbered 0 to 31
  - 32-bit data called a “word”
- **Assembler names**
  - $t0, $t1, …, $t9 for temporary values
  - $s0, $s1, …, $s7 for saved variables
- ***Design Principle 2:* Smaller is faster**
  - c.f. main memory: millions of locations

Register Operand Example

- **C code:**

```
f = (g + h) - (i + j);
```

For instance, f, …, j stored in $s0, …, $s4

- **Compiled MIPS code:**

```
add $t0, $s1, $s2
add $t1, $s3, $s4
sub $s0, $t0, $t1
```

Data Types

- Instructions are all 32 bits
- Byte(8 bits), halfword (2 bytes), word (4 bytes)
- A character requires 1 byte of storage
- An integer requires 1 word (4 bytes) of storage

Memory Operands

- **Main memory used for composite data**
  - Arrays, structures, dynamic data
- **To apply arithmetic operations**
  - Load values from memory into registers
  - Store result from register to memory
- **Memory is byte addressed**
  - Each address identifies an 8-bit byte
- **Words are aligned in memory**
  - Address must be a multiple of 4
- **MIPS is Big Endian**
  - Most-significant byte at least address of a word
  - *c.f.* Little Endian: least-significant byte at least address

Memory Operand Example #1

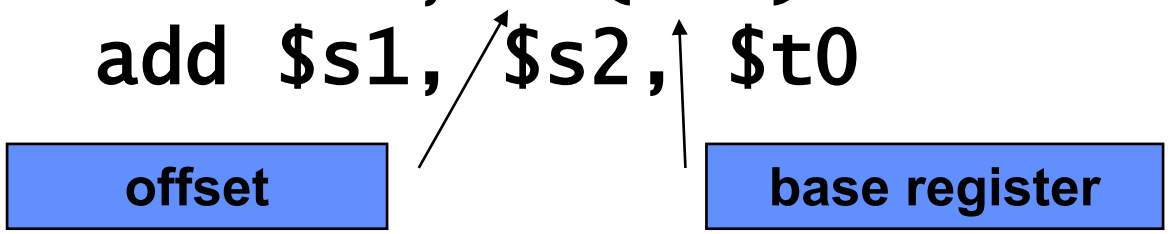
- **C code:**

```
g = h + A[8];
```

  - Assume that g is in $s1, h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  - Index 8 requires offset of 32
    - » 4 bytes per word

```
lw  $t0, 32($s3)    # load word
add $s1, $s2, $t0
```



Memory Operand Example #2

- **C code:**

```
A[12] = h + A[8];
```

  - h in $s2, base address of A in $s3

- **Compiled MIPS code:**
  - Index 8 requires offset of 32

```
lw  $t0, 32($s3)    # load word
add $t0, $s2, $t0
sw  $t0, 48($s3)    # store word
```

Registers vs. Memory

- **Registers are faster to access than memory**
- **Operating on memory data requires loads and stores**
  - More instructions to be executed
- **Compiler must use registers for variables as much as possible**
  - Only spill to memory for less frequently used variables
  - Register optimization is important!