18-447 Lecture 5: Instruction Set Architecture

James C. Hoe
Dept of ECE, CMU
February 2, 2009

Announcements: HW 1 due
Midterm in 2 weeks
Make sure you find lab partners for Lab2, no exceptions

Handouts: Handout04: Lab 2 (on Blackboard)
Handout05: HW2 (on Blackboard)
Handout06: HW1 Solutions (on Blackboard later)

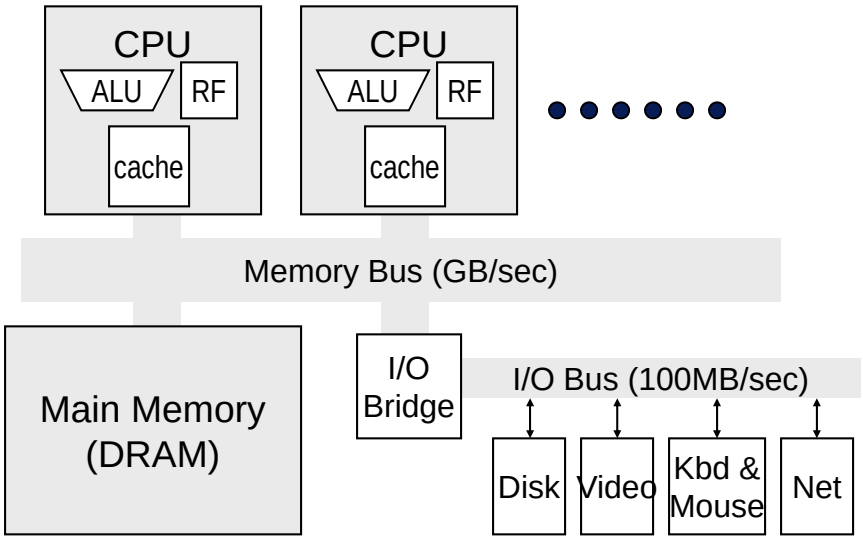
18-447: Road Map

15-213

- Computer Architecture
  - how to present precisely the functionality of a computer to a programmer
  - By the same token, this is also the most abstract "design spec" for the hardware guys
- Computer Organization
  - how to assemble
  - how to evaluate
  - how to tune
- Computation Structures
  - digital representations
  - processing, storage and I/O elements

18-240

How to specify what a computer does?



Architecture*

- "The term *architecture* is used here to describe the attributes of a system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flow and controls, the logical design, and the physical implementation."

  --- footnote on page 1, Architecture of the IBM System/360, Amdahl, Blaauw and Brooks, 1964.

How to specify what a computer does?

- Architectural Level
  -  A clock has a hour hand and a minute hand, ...
  -  A computer does ....????....

I know how to read a clock without knowing anything below

- Implementation Level
  -  A particular clock design has a certain set of gears arranged in a certain configuration
  -  A particular computer design has a certain datapath and a certain control logic
- Realization Level
  -  Machined alloy gears versus stamped sheet metal gears
  -  CMOS versus ECL versus vacuum tube

[Computer Architecture, Blaauw and Brooks, 1997]

Stored Program (von Neumann) Architecture

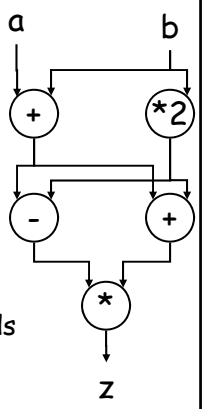
- stored program
  - instructions in a linear memory array
  - instructions can be modified just like data
- sequential instruction processing
  - program counter identifies the current instruction
  - instruction is fetched from memory and executed
  - program counter is advanced (according to instruction)
  - repeat

Burks, Goldstein, von Neumann, Preliminary discussion of the logical design of an electronic computing instrument,1946.

von Neumann vs Dataflow

- Consider a von Neumann program
  - What is the significance of the program order?
  - What is the significance of the storage locations?

```
v <= a + b;
w <= b * 2;
x <= v - w
y <= v + w
z <= x * y
```



- Instruction ordering specified by dataflow dependence (no program counter!!)
  - each instruction specifies who should receive results
  - an instruction can execute "whenever" all operands are received

Which one is more natural?

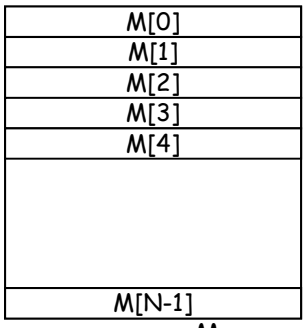
[figure and example from Arvind]

What are specified/decided in an ISA?

- Data format and size
  - character, binary, decimal, floating point, negatives
- "Programmer Visible State"
  - memory, registers, program counters, etc.
- Instructions: how to transform the programmer visible state?
  - what to perform and what to perform next
  - where are the operands
- Instruction-to-binary encoding
- How to interface with the outside world?
- Protection and privileged operations
- Software conventions

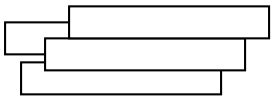
Very often you compromise immediate optimality for future scalability and compatibility

Programmer Visible State



Memory

array of storage locations indexed by an address



Registers

- given special names in the ISA (as opposed to addresses)
- general vs. special purpose

Program Counter

memory address of the current instruction

Instructions (and programs) specify how to transform the values of programmer visible state

General Instruction Classes

- Arithmetic and logical operations
  - fetch operands from specified locations
  - compute a result as a function of the operands
  - store result to a specified location
  - update PC to the next sequential instruction
- Data movement operations
  - fetch operands from specified locations
  - store operand values to specified locations
  - update PC to the next sequential instruction
- Control flow operations
  - fetch operands from specified locations
  - compute a branch condition and a target address
  - if "branch condition is true" then PC ← target address
    else PC ← next seq. instruction

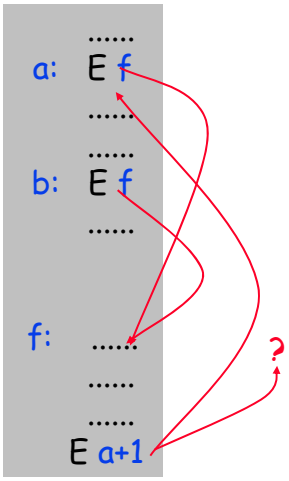
Generally defined to be atomic. Why?

An Early ISA: EDSAC

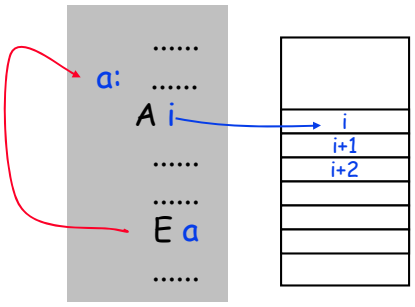
- Single accumulator architecture, i.e. ACC ⇐ ACC ⊕ M[n]
  - **A n**: add M[n] into ACC
  - **T n**: transfer the contents of ACC to M[n]
  - **E n**: If ACC>-1, branch to M[n] or proceed serially
  - **I n**: Read the next character from paper tape, and store it as the least significant 5 bits of M[n]
  - **Z**: Stop the machine and ring the warning bell
- Notice: only absolute addressing in data transfer and control transfer

Would you make the same mistake?

- Function call



- Array access in a loop



What is the fix?
What is the correct fix?

Evolution of Register Architecture

- Accumulator
  - a legacy from the "adding" machine days

    Ever wonder about that "AC" button on your calc?
- Accumulator + address registers
  - need register indirection
  - initially address registers were special-purpose,

    i.e., can only be loaded with an address for indirection
  - eventually arithmetic on addresses became supported
- General purpose registers (GPR)
  - all registers good for all purposes
  - grew from a few registers to 32 (common for RISC) to 128 in Intel Itanium

What are driving the changes?

Operand Sources?

- Number of Operands

Monadic	OP in2	(e.g. EDSAC)
Binatic	OP inout, in2	(e.g. IBM 360)

  - 3 operands in a smaller encoding to save memory

Triadic	OP out, in1, in2	(e.g. MIPS)

  Are there ISAs without (explicit) operands?
- Can ALU operands be in memory?

Yes!	e.g. x86/VAX/"CISC"
No!	e.g. MIPS/RISC/load-store architectures

- How many different variations

a very few	e.g. MIPS / RISC
a lot	e.g. x86
everything goes	e.g. VAX

Memory Addressing Modes

- Absolute LW rt, 10000
  use immediate value as address
- Register Indirect: LW rt, (r_{base})
  use GPR[r_{base}] as address
- Displaced or based: LW rt, offset(r_{base})
  use offset+GPR[r_{base}] as address
- Indexed: LW rt, (r_{base}, r_{index})
  use GPR[r_{base}]+GPR[r_{index}] as address
- Memory Indirect LW rt ((r_{base}))
  use value at M[GPR[r_{base}]] as address
- Auto inc/decrement LW Rt, (r_{base})
  use GRP[r_{base}] as address, but inc. or dec. GPR[r_{base}] each time
- Anything else can you think off .....

VAX-11: ISA in mid-life crisis

- First commercial 32-bit machine
  considered an important milestone
- Ultimate in "orthogonality" and "completeness"
  All of the above addressing modes x { 7 integer and 2 floating point formats} x {more than 300 opcodes}
- Opcode in excess
  - 2-operand and 3-operand versions of ALU ops
  - INS(/REM)QUE (for circular doubly-linked list)
  - "polyf": 4th-degree polynomial solve
- Encoding
  addl3 r1,737(r2),(r3)[r4] 7-byte, sequential decode

The first VAX11-780 was installed at CMU!!

MIPS RISC

- Simple operations
  - 2-input, 1-output arithmetic and logical operations
  - few alternatives for accomplishing the same thing
- Simple data movements
  - ALU ops are register-to-register (need a large register file)
  - "Load-store" architecture
- Simple branches
  - limited varieties of branch conditions and targets
- Simple instruction encoding
  - all instructions encoded in the same number of bits
  - only a few formats

Loosely speaking, an ISA intended for compilers rather than assembly programmers

Evolution of ISA

- Why were the earlier ISAs so simple? e.g. EDSAC
  - technology
  - inexperience, lack of precedence
- Why did it get so complicated later? e.g. VAX11
  - assembly programming
  - lack of memory size and performance
  - microprogrammed implementation
- Why did it became simple again? e.g. RISC
  - memory size and speed (cache!)
  - compilers
- Why x86 is still "king of the hill"?
  - technology vs. economics
  - technology vs. psychology
  - technology vs. deep pocket

CISC
Complex Instruction Set Architecture

Reduced Instruction Set Architecture

What if you had to design an ISA?

Open-Ended Design:
extrapolation and anticipation

"a dependable base for a decade of customer planning and customer programming, and continuing laboratory development…" Typical life expectance 15~20 years, but…..

- "Asynchronous" operation of components
  - abstract out exact time, performance etc to allow changing technology and relative speed of components
  - Note: this doesn't say to build them as asynchronous logic!
- Parameterization of storage capacity, multi CPU, multi I/O, etc
- Permit future extensions by "reserving" spare bits in instruction encoding
- Standard interfaces for expansion sub-systems

[Amdahl, Blaauw and Brooks, 1964]

General Purpose

- General Purpose = effective support for "large and small, separate and mixed applications" in many domains (e.g., commercial, scientific, real-time….)
- How
  - code-independent operation
    - no special interpretation of bit pattern in data

    e.g. ASCII character has no special significance per se
    - except where essential

    e.g., integer, floating point, etc
  - support full generality of logic manipulation on bit and data entities
  - fine-grain memory addressability (down to small units of bits)

[Amdahl, Blaauw and Brooks, 1964]

Inter-Model Compatibility

- Strict program compatibility = "a valid program whose logic will not depend implicitly upon time of execution and which runs upon configuration A, will also run on configuration B if the latter includes at least the required storage, at least the required I/O devices …."
- Invalid programs […] are not constrained to yield the same result
  - "invalid program" means a program that violates the architecture manual and not that it generates exceptions
  - exceptional conditions are part of the architecture
- By virtualization of logical structures and functions
- The King of Binary Compatibility: Intel x86, IBM 360
  - software base
  - performance scalability

[Amdahl, Blaauw and Brooks, 1964]

Wrap-up: Single-Instruction ISA

- What is the simplest single instruction ISA that is Turing-equivalent? *An academic curiosity*

- E.g.
  Subtract-Branch-If-Negative a1 a2 dest
  M[a1] ⇐ M[a1] – M[a2]
  if M[a1] < 0 then goto M[dest]
  else go to next

universality check list
conditional branch
load and store
universal logic function

Wrap-up: Terminologies

- Instruction Set Architecture
  - the machine behavior as observable and controllable by the programmer
- Instruction Set
  - the set of commands understood by the computer
- Machine Code
  - a collection of instructions encoded in binary format
  - directly consumable by the hardware
- Assembly Code
  - a collection of instructions expressed in "textual" format
    e.g. Add r1, r2, r3
  - converted to machine code by an assembler
  - one-to-one correspondence with machine code
    (mostly true: compound instructions, address labels ....)