CS 232 Computer Organization, Fall 2018

Dr. Bruce A. Maxwell
Department of Computer Science
Colby College

Course Description

Computer organization focuses on how computers work. Students learn the fundamental hardware components of computers, including storage (RAM, hard disks), input/output, and most importantly the processor (CPU). They learn how computer components are designed and built on several levels, including the basic electrical component level and the machine language level. They also learn to program in assembly language for one or more simple computer processors.

Prerequisites: CS 231

Desired Course Outcomes

A. Students understand standard binary encodings of data and programs.

B. Students understand the basic electronic components that make up a computer.

C. Students understand the computer at various layers, including the hardware layer, the machine language layer, and the assembly language layer.

D. Students are able to write assembly language programs for a simple CPU.

E. Students understand the significance of new technology in computer science.

F. Students present methods, algorithms, results, and designs in an organized and competently written manner.

1 Introduction

Introduction sequence: where computer architecture/organization fits

Administrivia

- Course home page
- Labs due Monday evening
- One free delay of the deadline to Friday (not labs that finish on Friday)
- homework (given Wed, due Friday)
- Quizzes (almost every Friday except this week)
- Textbook: there is a textbook (yes, it will feel a bit old), HW will include questions from the textbook

Moore's Law (1965): twice as many transistors on a chip every 2 years

- What is a transistor?
- Show how you could use a switch to invert a signal (Boolean logic operation NOT)
- Moore's law plot
- Current size of a transistor: 14nm (Si atom is 0.2nm, what is 14/0.2?)
- How do we design chips with 100s of billions of transistors?

Design tools

- Hierarchical design
- Design languages
- Simulation environments
- Compilers to move between levels

Why architecture is important: go through the two examples

- Python example using 4.35 and int(100 * 4.35)
- C example using an int and incrementing in an infinite loop

Binary encoding

- Counting
- unsigned binary
- box notation

2 Boolean Concepts

Binary encodings

- unsigned binary numbers
- Important numbers: byte/kilo/mega/giga/terra/peta
- Fixed size representations: wrapping
- signed binary
- 2's complement

Representations other than binary: Octal, Hex, ASCII

Boolean Logic

- math with Boolean values (0/1)
- standard operators: AND, OR, NOT
- AND is the equivalent of multiplication (0 * X is 0, 1 * X is X)
- OR is the formal equivalent of addition (0 + X is X, 1 + X is 1)
- A Boolean expression is an algebraic expression that evaluates to a 0 or a 1

Truth tables

- Another way to represent a Boolean expression
- Write out all possible input values
- Write out the corresponding output value
- Each line of the truth table corresponds to a Boolean AND expression
- The set of expressions that correspond to 1s in the output define a Boolean expression for the circuit

3 Combinational Circuit Design

*****Look at a real computer

Logic circuits from Boolean expressions

- symbols for NOT, AND, OR, NAND, NOR, XOR, BUFFER
- Look at the set of expressions corresponding to ones
- AND-OR tree

Motivate combinational circuit design with an adder

- Half-adder: two inputs, two outputs (sum, carry)
- Full-adder: three inputs, two outputs (sum, carry)

Truth Tables

Identifying circuits directly from the 1 terms of the truth table (or the 0 terms)

Optimizing Circuits

Boolean Axioms (not the way we optimize circuits)

Karnaugh Maps (the way we optimize small circuits of 6 variables or less)

Design process

- Truth table
- Karnaugh map: group 1's by connected blocks of powers of 2
- Write the Boolean expression
- Draw the logic diagram
- Build the circuit

Examples:

ABC: 00 / 10 / 11/ 11

ABC: 10 / 01 / 10 / 01

ABCD: 0000 / 1111 / 0101/ 1101

4 Combinational Circuits II

Review Karnaugh Maps (the way we optimize small circuits of 6 variables or less)

Design process

- Truth table
- Karnaugh map: group 1's by connected blocks of powers of 2
- Write the Boolean expression
- Draw the logic diagram
- Build the circuit

Take a look at the sum circuit (XOR)

One more example: ABCD: 1010 / 0000 / 1010 / 1111

Standard form: sum of products

AND-OR tree maps to NAND-NAND tree

Building a gate from transistors: P-type (current flows when low), N-type (current flows when high)

- NOT gate: two transistors in series with the output in the middle
- NOR gate: two p-type on top in series, two n-type on the bottom in parallel
- NAND gate: two p-type on top in parallel, two n-type on the bottom in series
- OR gate: NOR gate with a NOT gate (6 transistors)
- AND gate: NAND gate with a NOT gate (6 transistors)

Useful combinational circuits

- adder
- multiplexer [MUX]: routes one of 2^N inputs to one output

5 Combinational Circuits III

Homework:

- -5 is -8 + 3, so $-5 = 1011$
- Add 0011 and 1010. You get 1101, or -8 + 5 = -3, with a carry from the second position
- The K-map has four groups: one singleton, two groups of two, and one group of four
- $A\bar{B}C\bar{D} + \bar{A}B\bar{D} + \bar{A}\bar{B}D + B\bar{C} = F$

Project Comments

- Counter: Megafunctions::arithmetic::lpm_counter, you want a 4-bit up counter with an asynchronous clear, no other options
- Don't use the 35MHz clock to drive the counter, use a push button
- If you use a push-button as a reset value, send it through a NOT gate: buttons are default high
- use labeling to connect wires; use indexing to select bits from a bus (define bus v. wire)

More Useful combinational circuits

- review MUX: routes 2^N inputs to one output (correct from Wed)
- demultiplexer [DEMUX]: routes an input signal to one of 2^N outputs, may have an enable input
- decoder: sets one of 2^N outputs to high
- priority encoder [ENC]: output is a number indicating which of 2^N input lines is high, second output indicating all are low.
- comparator: output is high when two numbers are identical (XOR pattern)

Example: Implementing a truth table as a demux circuit.

Quiz (20min)

6 VHDL Basics

Useful combinational circuits (finish)

- decoder: sets one of 2^N outputs to high
- priority encoder [ENC]: output is a number indicating which of 2^N input lines is high, second output indicating all are low.
- comparator: output is high when two numbers are identical (XOR pattern)

Library: importing useful packages

- library ieee;
- use ieee.std_logic_1164.all;
- use ieee.numeric_std.all;

Entity: port statement as the function definition

- entity name has to match the filename (thing.vhd)
- entity almost always consists of a single port statement

Signals: IEEE standard useful types

- a: in std_logic;
- b: out std_logic_vector (3 downto 0); – comment
- c: in unsigned (7 downto 0);

Architecture: declarations and code

- Need to name the architecture definition with a unique name (no special requirements)
- Can create multiple architecture definitions (generally for large projects)

Declaration section (after architecture but before the begin)

- Temporary local variables and internal memory/state
- Component declarations: defining other circuits to be included in this circuit

Code section (between begin and end)

- Concurrent statements: all statements in a VHDL architecture act simultaneously

Timed assignments: useful for creating (small) test circuits: a ¡= ’0’, ’1’ after 5 ns, ’0’ after 10 ns;

Conditional signal assignments / select statements (DEMUX)

- a <= ’1’ when b = ’0’ else ’1’; – note single v. double quotes, can build a truth table
- with a select b <= ’1’ when ’0’, ’0’ when others;

Component instantiation

- label: <component> port map(<arguments>)

7 Sequential Circuits

Current events questions for Friday

- What are the top 3 supercomputers?
- How fast are they?
- Who was Seymour Cray?

VHDL Tips

Conditional Signal Assignments

```
F <= "00" when b = "000" or b = "101" else "01" when b = "001" else "10" when b = "010" or
```

Digital Memory

Flip-flops

- NOR-NOR latch
- Gated D-latch: NOR-NOR with AND gates as input, D (top) and $\bar{D}$ bottom
- Flip-flop: Rising edge memory circuit: put an AND gate with A and $\bar{A}$ going to it (path-lengths)
- Symbol for a D F/F: D / Clk / Q / asynchronous reset

Useful Sequential Circuits

Registers

8 Sequential Circuits

HW

1. The number is multiplied by $2^3 = 8$. 000011 = 3 becomes 011000 = 24, and 000110 = 6 becomes 110000 = 48.
2. Using a 4-1 MUX you can use two columns of inputs as the control signals. For three input signals, that makes each input of the MUX correspond to two lines of the truth table. The input function must be one of 1, 0, C, or $\bar{C}$. In this case, the AB = 00 input is 1, the AB = 01 input is C, the AB = 10 input is 0, and the AB = 11 input is $\bar{C}$
3. The following Python code snippets represent each function.

```
def MUX( I, C ):
  if C = "00": return I[0]
  elif C = "01": return I[1]
  elif C = "10": return I[2]
  else: return I[3]
```

```
def DEMUX(A, C):
  if C = "00": return [A, 0, 0, 0]
  elif C = "01": return [0, A, 0, 0]
  elif C = "10": return [0, 0, A, 0]
  else: return [0, 0, 0, A]
```

```
def DEC(C):
  if C = "00": return [1, 0, 0, 0]
  elif C = "01": return [0, 1, 0, 0]
  elif C = "10": return [0, 0, 1, 0]
  else: return [0, 0, 0, 1]
```

```
def PENC( I ):
  if I[0] = 1: return ("00", 0)
  elif I[1] = 1: return ("01", 0)
  elif I[2] = 1: return ("10", 0)
  elif I[3] = 1: return ("11", 0)
  else: return ("11", 1)
```

Quiz

9 State Machines

Quiz Review

Useful Sequential Circuits

Shifters: moving bits left and right

- implements math
- consider each bit as having a 4-1 MUX attached: load, hold shift left, shift right

Counter

- N D flip-flops
- Adder (can be optimized)
- Toggle for up-down
- Synchronous or asynchronous clear
- Load option
- use a 4-1 MUX for each bit: load, hold, count up, count down

State Machines

- Definition: describes a process
- Example: open door/closed door, NPC in a computer game
- State machine circuits have three parts: state memory, next state logic, output logic
- Number of states in the state machine determines the number of memory bits required
- Moore Machine: output is a function of the state
- Mealy Machine: output is a function of the state and the input (can change at arbitrary times)

VHDL State Machines

```
library ieee;
use ieee.std_logic_1164.all;

entity four_state_moore_state_machine is
   port( clk : in std_logic; input  : in std_logic; reset  : in std_logic;
         output : out std_logic_vector(1 downto 0) );
end entity;

architecture rtl of four_state_moore_state_machine is
  type state_type is (s0, s1, s2, s3);
  signal state   : state_type;
begin
  process (clk, reset)
  begin
    if reset = '1' then
      state <= s0;
    elsif (rising_edge(clk)) then
      case state is
      when s0 => if input = '1' then state <= s1; else state <= s0; end if;
      when s1 => if input = '1' then state <= s2; else state <= s1; end if;
      when s2 => if input = '1' then state <= s3; else state <= s2; end if;
      when s3 => if input = '1' then state <= s0; else state <= s3; end if;
      end case;
    end if;
  end process;

  process (state)
  begin
    case state is
      when s0 => output <= "00";
      when s1 => output <= "01";
      when s2 => output <= "10";
      when s3 => output <= "11";
    end case;
  end process;
end rtl;
```

10 Building Blocks

Register File

- Set of addressable registers
- Decoder activates one register's write-enable signal
- Overall control signal controls if any register should load
- All registers are connected to the same input bus
- MUX connects all registers to an output bus
- Control signal enables the connection to the output bus
- High Impedance: when an output is basically disconnected

General-purpose ALU

- Useful functions: Add, Sub, SLL, SLR, SAR, ROR, ROL, Negate, Multiply, Divide
- Shifting inputs using a MUX
- Inverting 2's complement to implement subtraction (flip bits and add one)
- Detecting overflow, carry-out, carry-in, sign bit
- Flag register: indicates the status of the last ALU operation

Bus

- A set of wires that carry a signal from one set of devices to another.
- Only one device at a time can use the bus
- A design component that helps minimize physical routing pathways

ROM: Read-only Memory

- A truth table
- Can be implemented as a multi-bit MUX
- In practice, implemented as a matrix of wires or write-once memory

Instruction Register

- Holds the current set of control signals
- ALU op, Bus source and destinations, control flow modifications

Program Counter

- Holds the address of the next instruction
- Incorporates increment logic
- Can load any arbitrary address

11 Programmable Circuit

HW Review

Example

- Program Counter and Instruction Register
- two registers A and B
- ALU with Add, PASS input B (1 control bit)
- ALU Sources: RA and one of {RB, 1 0, -1} (2 control bits)
- ALU Destination: RA or RB (1 control bit)

Code: The Count

- RA and RB have 0's to start (on reset)
- Program counter is at 0 to start (on reset)
- 8-line program that repeats
- I0: I0I1 = 01 C0 = 0 O0 = 0 : RA <= RA + 1 : Increment RA
- I1: I0I1 = 10 C0 = 0 O0 = 1 : RB <= RA + 0 : Move RA to RB
- I2: I0I1 = 10 C0 = 1 O0 = 0 : RA <= 0 : Move 0 to RA
- I3: I0I1 = 11 C0 = 1 O0 = 0 : RA <= -1 : Move -1 to RA
- I4: I0I1 = 10 C0 = 1 O0 = 0 : RA <= 0 : Move 0 to RA
- I5: I0I1 = 00 C0 = 0 O0 = 0 : RA <= 0 : Add RA to RB and put it in RA

An Instruction

- The set of control bits required to tell the circuit what to do.
- The instruction set is the space of all possible control signals

How do we modify our machine to be able to do control flow?

Quiz

12 A CPU

Current Events Questions

Who were Alan Turing and John Von Neumann?

What is the largest hard drive you can get for $100?

What is the largest solid state drive you can get for $100?

Extending *The Count*

- Requires the ability to modify the program counter [PC]
- Need a new MUX for input to the PC: sources are the IR bits or PC + 1
- Requires some condition to test to know if the PC needs to load a new value
- Need a Condition Register [CR] to hold the condition of the ALU
- Probably could make IR data bits an input to ALU bus B instead of 1
- Need both unconditional and conditional jumps to implement a for loop
- Requires an additional clock cycle to load the new PC value
- Requires some state logic to implement the F/E cycle
- Write enable on RA/RB/PC/CR is E cycle, Write enable on IR is F cycle

13 A Better CPU

Quiz Review

A for loop for The Count

```
0 LOAD RB  0
1 LOAD RA  5
2 ADD  RA RB RB
3 ADD  RA -1 RA
4 BR0  6
5 BR   2
6 MOV  RB RA
```

New ALU capabilities: and / or / xor / not

Questions:

- How many bits is each instruction?
- Does each instruction use all of the bits?
- How many possible instructions could we have?
- Do we have to organize the bits in any particular manner?
- Does it make sense that we have organized them this way?
- What could we name the fields?
- How could we classify the instructions in terms of the 4 things a computer can do?

HW Question: How could you swap RA and RB with the new set of instructions?

14 Binary Jeopardy

Discussion of homework

Quiz #4

Speed round 1: binary, hex, and decimal numbers, 1 min per pair (do this as just hex/binary in the future)

Speed round 2: powers of 2, 1min per pair

Final round: every pair write down their answer (didn't get to this)

Questions:

- Who was the first computer programmer?
- Who developed a test for whether machines are intelligent?
- Who built the first supercomputer?
- Who founded Intel?
- What is Moore's Law?
- What is the size of a transistor feature in a current CPU?
- In what country is the world's fastest supercomputer according to top500.org?
- What is the largest hard drive you can buy for $100?
- What is a FLOP?
- How many operations (in decimal?) per second can a computer capable of 122 petaFLOPs execute?
- Who designed and tried to build the first general purpose computing machine?
- Who led the effort to break the German ENIGMA code in WWII?
- Who came up with the basic computer architecture concept called the stored memory computer?

15 The First CPU

Current events: what is the cost of 1GB of main memory?

Store Data: Registers and memory

Move Data: Buses and Multiplexers

Manipulate Data: ALU

Adjust Control Flow Based on Data: Control logic

ENIAC: Eckert and Mauchly (Von Neumann as a consultant)

- Vacuum tubes, 30 tones, 15,000 ft^2, 140kW, 5000 additions/s, operated until 1955

The IAS Computer (1952)

Designed by John Von Neumann

- Primary novel concept was storing the program in memory (otherwise had to use cabling on ENIAC)
- 4096 memory locations of 40 bits each
- Each word could hold one piece of data or two instructions
- Instructions contained an 8-bit opcode and a 12-bit address (1024 locations)

Memory

Overall design:

- DRAM: dynamic RAM
- SRAM: static RAM
- MRAM: magnetic RAM
- ROM (solid state memory / one-time)

[Definitions and Characteristics of Computer Memory

- Big v. Little Endian
- Location: CPU, on chip, off-chip, way off chip
- Capacity
- Cost per bit
- Unit of transfer
- Access method: sequential, direct, random, associative
- Addressable Units
- Performance: access time, cycle time, transfer rate, peak transfer rate
- Physical type: semiconductor, magnetic surface, optical surface
- Physical characteristics: volatile/nonvolatile, erasable/nonerasable

16 Memory

Current cost:
$10 per GB of DDR4 (double-data rate) Non-ECC (error-correcting code) 2400MHz S(synchronous) DRAM

Endian-ness: big-endian v. little-endian

Memory design: A decoder for the rows, a MUX for the columns, split the address into two parts

RAM timing patterns

Reading data:

- Address decoding
- Chip select
- Output Enable
- Output Data

Writing data:

- Address decoding
- Data setup time
- RW signal: set low to indicate a write, rising edge captures the data

Synchronous v. asynchronous memories: synchronous memories use a clock

- Synchronous can be faster because you can add a state machine and memory
- Pipeline: stages to a process, allow multiple stages to be running simultaneously

Memory Hierarchy Design

- Two levels: registers + main memory
- Three levels: registers + L1 cache + main memory
- Four levels: registers + L1 cache* + L2 cache + main memory
- Five levels: registers + L1 cache* + L2 cache* + L3 cache + main memory

17 Cache

What is your L1/L2/L3 cache?

Memory Hierarchy Design

- Two levels: registers + main memory
- Three levels: registers + L1 cache + main memory
- Four levels: registers + L1 cache* + L2 cache + main memory
- Five levels: registers + L1 cache* + L2 cache* + L3 cache + main memory

Locality of reference

Performance: Ts = T1 + (1 - H)*T2, T1 = cache speed, T2 = main memory speed, H = Hit rate

Cost per bit: weighted average of cache and main memory

Cache Design

Cache organization: cache lines

- Cache lines are small enough to be transferred quickly
- Cache lines are long enough to bring across high probability future accesses
- Cache lines are short enough to minimize transfers without purpose.
- Cache lines are always powers of 2 in length

Mapping Functions: to what line of cache does a line of memory map?

- Direct mapping: modulo function, tag field indicates source of the line
- Tag is the part of the address we need to remember
- Line field: what part of the address corresponds to the line?
- Byte field: which byte on the cache line?

Diagram out the bits of an address

18 Cache II

Review: Cache is organized in lines

- Cache lines are small enough to be transferred quickly
- Cache lines are long enough to bring across high probability future accesses
- Cache lines are short enough to minimize transfers without purpose.
- Cache lines are always powers of 2 in length

Mapping Functions: to what line of cache does a line of memory map?

- Direct mapping: modulo function, tag field indicates source of the line **(review)**
- Associative mapping: any line, tag field is the source of the line **(review)**
- Set associative mapping: v sets, k lines, modulo v to pick the set (new)

Example: 32-bit addresses, 1G (2^{30}) Main memory, 64K (2^{16}) cache with 16 byte cache lines (2^4), 4K (2^{12}) cache lines.

- Direct mapping: 32 bit address modulo number of cache lines, means 2^{20} lines in main memory map to each cache line.
- 2-way Set Associative: 32 bit address modulo number of sets, means 2^{21} lines in main memory map to each pair of cache lines
- 4-way Set Associative: 32 bit address modulo number of sets, means 2^{22} lines

Replacement Algorithms:

- Direct mapping: only one option, trash what is there
- Associative/Set associative mapping: least-recently used, use a bit for 2-way SA
- Approximate LRU: binary tree with 1 bit at each branch indicating most recent usage
- FIFO: round-robin using a circular buffer indicating the last line filled
- Least frequently used: use a counter at each line

Write policy:

- Write-through: all writes go straight to memory
- Write-back: set a dirty bit in the cache line, when the line is replaced send its contents to memory

Number of caches, other caches that exist in a computer system

Multi-processor caches: think about cache coherence

19 Cache III

Homework

- Question 1- tag: 17 set: 8 word: 7
- Question 2- Given 2^6 words/line and an 8k (2^{13}) byte cache, there are $128 = 2^7$ lines and $32 = 2^5$ sets for a 4-way SA cache. tag: 22 set: 5 word:6
- Question 3- tag: 18 line: 8 word: 6, so the bits in the line field for FF0110A8 are 01 0000 10 = 66 = 0x42.
- Question 4- Instructions are generally access in sequential order, so the likelihood that two instructions that would clash would be repeatedly accessed are small. Data accesses, however, are less sequential, so spending resources on reducing clashes, thereby increasing the hit rate, is worth the cost.
- Question 5- Without the L2 cache, 30% of the accesses would go to main memory. With the L2 cache, 6% of the accesses go to main memory.
- Question 6- tag: 24 set: 6 byte: 6
- Question 7- tag: 21 set: 9 byte: 6
- Question 8- tag: 17 set:13 byte: 6
- Question 9- you need 15 bits to make a binary tree subdivision of 16 lines

Unified v. Split cache

- Von Neumann architecture: unified memory for programs and data
- Harvard architecture: separate memories for programs and data
- Hybrid architectures: lowest level is a Harvard architecture, higher levels are Von Neumann

Principle of inclusion: a higher cache (closer to main memory) must contain everything in all lower caches. Otherwise, coherence is hard to maintain, as any access has to be queried all the way down.

Number of caches, other caches that exist in a computer system (optional)

- L-1 cache, on modern CPUs each core has its own, usually split between instruction and data
- L-2 cache, on modern CPUs each core has its own, unified for instructions/data
- L-3 cache, on modern CPUs the cores share a single unified cache
- disk cache: solid state memory as part of a disk drive
- main memory as cache for virtual memory (on the hard drive)
- caching for web pages: really annoying during development

Up next: Instruction Set Architecture

Quiz

20 Instruction Set Architecture

Quiz review

ISA: what is it?

- Registers: organization, number, size, type, addressability
- Connections between components: data motion
- ALU capabilities
- I/O capabilities
- Memory access capabilities: addressing memory
- Control flow options

The ISA does not necessarily imply a specific computer organization. Many different computer organizations can implement the same ISA. Example: IA-32 instruction set in use, with variations and expansions, since the 8086/8088, extended to 64-bits with x86-64.

Bus Organization is an overall driver of design

- Stack-based (e.g. Java Virtual Machine)
- One-bus/Two-bus (IA-32 is sometimes 2-bus)
- Three-bus with memory operands (IA-32/64)
- Three-bus with register operands (PowerPC / IA-32 extensions)
- Four-address ISAs

21 ISA: Instruction Types

CISC v. RISC

- CISC: complex instruction set CPU
- RISC: reduced instruction set CPU
- Size of program code v. complexity of the hardware
- RISC machines could run each instruction faster
- CISC machines required fewer instructions
- Current CPUs are CISC on RISC with hardware interpretation
- RISC is generally lower power; most embedded systems use a RISC architecture

IA-32: example of a CISC ISA

- Thousands of different instructions
- Variable length instructions
- Multiple memory addresses allowed
- Full size immediate values allowed
- Memory and register access allowed in a single instruction
- Some instructions execute extended operations (e.g. string copy)

PowerPC: example of a RISC ISA

- Fixed length 32-bit instructions
- All binary operations use registers
- Dedicated load/store operations
- First six bits are the opcode, for some ops, other bits provide additional information

Types:

Instructions

What are common instructions types?

- Data transfer
- Arithmetic and Logical operations
- Control
- System control

22 HW Review

Homework Review

1. How many main instructions can the PowerPC instruction set contain? 6-bit main opcode, $2^6 = 64$

2. How many bits of the opcode (instruction) are necessary to specify the two operands and the destination for an ADD operation? Given the 6-bit main opcode, how many bits are left to provide other information?

5 bits per register, 6-bits + 15-bits, leaves 11 bits for other information

3. What can you do with those bits? Additional opcode, immediate values, set condition register, handle types, mask operands

4. The unconditional branch instruction has 6 bits of opcode, followed by 24 bits of address, sign-extended. Bit 30 of the opcode specifies whether the address is to be treated as an absolute address or relative to the current value of the PC. In the case of an absolute address, what parts of memory could it reach?

The top 2^{23} of memory and the bottom 2^{23} of memory.

5. There are 21 different LOAD instructions, and 17 different STORE instructions. What percentage of the total possible instructions do these instructions constitute? Why commit that many instructions to LOAD and STORE operations?

All data must be moved in or out of memory to registers in order to be manipulated. With a fixed size address, you have to get creative how you can address parts of memory. In addition the instructions allow loading different amounts of data.

Addressing Memory: What kind of variety is there?

- Immediate: value is in the instruction
- Direct: address is in the instruction
- Register: value is in the register
- Register Indirect: value is in the address specified by the register
- And many more...

23 ISA: Control Instructions

Current Events Question: what are some current events in hardware design? Margaret Heafield Hamilton, Grace Hopper

Instruction Types

- Data transfer (review, remember PUSH/POP, system stack)
- Arithmetic and Logical operations (review)
- Control
  - Branch, Conditional branch
  - Skip, Skip conditional
  - Halt, Wait (condition)
  - No-op
  - Execute: execute one instruction at the specified address
  - Call: jump to a subroutine or function, store PC
  - Return: return from a subroutine or function, restore PC

    Function call procedure:
    - Push space for return value(s)
    - Push arguments
    - CALL: pushes return address on the stack
    - Push old stack frame
    - Set the stack frame to the stack pointer
    - Push local variables, if necessary
    - Push the values of any registers to be used
    - Execute the subroutine
    - Pop the values of any registers used
    - Pop the local variables, if any
    - Pop the old stack frame and restore it
    - RETURN: pops the return address to the PC
    - Pop the arguments
    - The top of the stack is the return value

    Ways of passing arguments: put the values on the stack, put the values in registers: hardware support can make this fast.
  - Interrupt: INT transfer control to an interrupt handler
  - Return from Interrupt: RFI return from an interrupt

24 ISA: Addressing

Current Events

- Margaret Hamilton, Grace Hopper
- Current events: real time ray tracing GPUs
- Red Dead Redemption II: $750M in 3 days

Addressing Memory

- Immediate: value is in the instruction
- Direct: address is in the instruction
- Indirect: operand is in the address specified by the address in the instruction
- Register: value is in the register
- Register Indirect: value is in the address specified by the register
- Displacement:
  - Relative addressing: address is immediate displacement + PC
  - Base register addressing: address is immediate displacement + register
  - Indexed addressing: address is immediate address + index register
  - Register indexed addressing: address is base register + index register
  - Scaled Indexed Base addressing: address base register + index register * scale
  - Preindexing: get the value indicated by the direct address in the instruction and add it to an index register
  - Postindexing: add the value in the instruction to a register to get the memory location that contains the address
  - Stack: operand(s) is(are) at the top of the stack, stack register holds the address

25 ISA: Registers

HW Review

Registers, Types, and Organization

Types

- Integer (8/16/32/64 bits in size)
- Address
- Floating point

Organization: register file, stack, special v. general, address v. data

- General purpose: used for addresses, data
- Data: cannot be used as part of an addressing mode
- Address: often specifically used to hold addresses for specific instructions
- Condition: hold condition codes

Design considerations:

- All general purpose, or lots of specialized registers?
- Number of registers (CISC v. RISC and addressing modes)
- More registers is more access/decode time
- Register length

Control & Status registers

- PC / IR / MAR / MBR
- Program status word: condition codes plus other status information

Quiz

26 ISA: Register Designs

Current Events: Who are Kathleen Booth, John Backus, and Dennis Ritchie?

Pentium

- General: 8 x 64-bit registers (addressable as 64/32/16)
- AX-DX, SP (stack pointer), BP (base pointer), SI (source), DI (destination)
- Some instructions use specific registers: string ops use SI, DI
- SP and BP are general-purpose, but used by convention
- 6 Segment registers: address base pointers
- Flags register: all condition flags ; Instruction Pointer [PC]
- 8 x 80-bit floating point registers, organized as a stack
- MMX/SSE registers: floating point registers used for SIMD instructions
- Interrupt Vector Table: 256 addresses, 0-31 are reserved

PowerPC

- 32 x 64-bit general purpose registers, GPR1 is used as stack pointer by convention
- 32 x 64-bit FP registers
- 8 x 4-bit condition codes (32-bit register), compare instruction sets any of the 4-bit chunks
- Link register: loaded with an address after branching
- Count register: can be used to control an iteration loop
- Machine State register: 64 bit register

PIC Microcontroller

- AC for all binary operations; 192 8-bit registers for holding data
- one indirect register for indirect addressing
- I/O register for each unit; PC, IR, Flags

ARM

- R0-R3: parameters to a procedure being called
- R4-R11: local variables, GPRs
- R12: Intraprocedure call register for 32-bit function calls
- R13: Stack Pointer
- R14: Link register, return address for current function
- R15: program counter, also has a PSW with zero, negative, overflow (and others)
- 32 x 32-bit floating point registers, or 16 64-bit doubles

27 ISA: Interrupts and Opcodes

Review: function calls

CALL: stores the return address, may do other things
RETURN: restores the return address to the PC, may do other things

Connect to register design:

- passing arguments using the stack: review process
- passing arguments using registers: ARM uses registers by convention
- hardware supported register-based argument passing: SPARC and rotating register files
- hardware supported stacks: keeping the top elements of a stack in registers
- return address register stack: fixed size stack for return addresses
- link register: single register for holding a return address, might have connection to a stack

Interrupts

How does a CPU handle asynchronous events: poll or handle some other way
Needs to be invisible to the program being interrupted

INT: software interrupt

- Push return address on the stack
- Push condition register / program status word on the stack
- Put the interrupt routine address in the PC

RTI: return from interrupt

- Pop the condition register / program status word
- Pop the return address to the PC

28 Assembly

Assembly

Assembly: one-to-one mapping between assembly instructions and machine instructions

- Label, operation, one or more operand fields
- Names for key components of the ISA (registers, I/O ports)

Pseudo-instructions: commands to the assembler

- storage, function definitions, conditional assembly, macros

Two-pass assembler

- First pass counts lines and creates a dictionary of labels (symbol table)
- Second pass creates the machine instructions

Tokenization: splitting up an input into atomic tokens stored in an array or list

Symbol table can include location labels, constants, and macros (if supported)

Opcode table contains an entry for each instruction type indicating its hex values

- May also contain a recipe for interpreting the rest of the assembly input
- May also contain a recipe for building the instruction

Assemblers sometimes contain instructions that don't exist in the machine language and have to be manufactured from several instructions.

- Load Immediate: translate into pre-storing a value in memory and loading that value

You can put assembly into C code. Show MMX instructions.

Quiz

29 Real Instruction Formats

Quiz Review: primarily the stack, but also remind CISC v. RISC

Two-pass assembler

- First pass counts lines and creates a dictionary of labels (symbol table)
- Second pass creates the machine instructions

Tokenization: splitting up an input into atomic tokens stored in an array or list
Use split, identify a label by its last character (colon)

Symbol table can include location labels, constants, and macros (if supported)
Dictionary: key is the label, value is the corresponding line number

Opcode table contains an entry for each instruction type indicating its hex values
Dictionaries: use lots of dictionaries to translate from mnemonics to machine code
The tables are there in lab 7.

Instruction Format Design Space

- Fixed or variable size
- Program size (more or fewer instructions)
- Instruction fetch speed
- Instruction decode speed
- Memory address size and addressable unit
- Richness and flexibility (is there space for expansion?)
- Bus size

Allocation of bits

- Number of addressing modes
- Number of operands
- Register v. memory access for alu ops
- Number of registers / register sets
- Address range (esp jumps) / addressable unit
- Expanding opcodes

Real instruction formats

PDP-8: 12-bit fixed length instructions

- only one register: don't need to specify the destination or source of operations
- 3 bits of opcode: instructions 0-7
- instructions 0-5 are memory reference instructions
  - Memory is divided into 128 byte pages, total accessible memory is 12 bits (4096)
  - Addressing can be direct/indirect (1 bit)
  - Addressing can be zero-reference or current page reference (1 bit)
  - 7 address bits in the instruction
  - 8 memory addresses in the first page will auto-index if used for indirect addressing
- instruction 6 handles I/O instructions, one of 64 I/O registers and 3 additional bits of opcode
- instruction 7 interprets the remaining bits as single bit micro-instructions which can be combined
- instruction 7 includes all branching instructions

PDP-10: 36-bit fixed length

- Orthogonality: address computation is independent of the operation
- Completeness: all operations are available for all data types
- Direct addressing: instruction can contain a full address
- Indirect addressing: addresses can be indirect and can use an index register
- 16 GP registers: 4 bits to specify
- GP registers can be used as index registers for array addressing
- Most operations use one register and one memory location
- 18 bit immediate value or address
- A single format: 9 bits opcode, 4 bits register, direct/indirect bit, index register, 18-bit immediate or address

30 Real Instruction Formats II

PDP-11: variable-length instruction, including 0, 1, 2-address instructions (CISC)

- One of the ultimate CISC instruction set
- Orthogonal: any addressing mode with any operand
- 8 GP registers, including stack pointer and PC
- 4 FP registers
- 6 bits to access one of 8 registers: extra 3 bits specify the addressing mode
- For FP instructions: 2 bits to specify a register
- thirteen different instruction formats, including 0-, 1-, and 2-address formats
- opcode is 4 to 16 bits long
- 16, 32, and 48 bit instructions are possible

IA-32(64): variable-length instruction set (CISC)

- Prefix (0-4 bytes)
  - Instruction prefix: LOCK/repeat - shared memory access / string ending conditions
  - Segment override - segment register is specified in the prefix
  - Address size - 16 or 32-bit addressing, one is default
  - Operand size - 16 or 32 bits, one is default
- 1-2 byte opcode
  - may include bits to specify 16 or 32 bit data
  - direction of the data operation (to or from memory)
  - whether immediate field should be sign-extended
- Mod r/m byte
  - Mod - combines with r/m field to create 32 values: 8 registers, 24 indexing modes
  - Reg/Opcode - holds either the address register (8) or 3 bits of opcode
  - r/m - specifies an operand register or part of the indexing mode
- SIBbyte
  - SS - scale factor for indexing (2 bits)
  - Index - index register (3 bits)
  - Base - base register (3 bits)
- Displacement (0, 1, 2, or 4 bytes)
- Immediate (0, 1, 2, or 4 bytes)

31 Real Instruction Formats III

HW Review

- GP v. FP instructions
- Maximizing the number of 3-address instructions
- Additional formats for addresses create more design problems

PowerPC: 32-bit fixed length RISC: 32-bit fixed size instructions, first 6 bits are primary opcode

- Branch Instructions (3 formats)
  - Branch: 24 bit immediate, absolute/relative flag, link to subroutine flag
  - Cond Branch: Options (5 bits), CR bit (5 bits), 14-bit displacement, absolute/relative flag, link
  - Cond Branch: Options (5 bits), CR bit (5 bits),
- Condition register instructions (1 format)
  - Destination bit specifier (5 bits)
  - Source 1 bit specifier (5 bits)
  - Source 2 bit specifier (5 bits)
  - 10 bits to specify the operation (Add, OR, XOR, AND, etc.)
- Load/Store instructions (3 formats, 1 64-bit only)
  - Destination/Source register (5 bits)
  - Base Register (5 bits)
  - Option 1: 16-bit Displacement
  - Option 2: Index register, opcode for size, sign, update
  - Option 3: 14-bit Displacement, 4 bits of additional opcode (64-bit)
- Integer arithmetic, shift/rotate instructions (9 formats, 3 64-bit only)
  - Destination register
  - Source register
  - Second source register or a shift/rotate amount, or part of a 16-bit S/U immediate value
  - Mask bits, additional opcode, or part of an immediate value (signed/unsigned 16-bit)
  - Set condition register flag
- Floating point instructions (1 format, two primary opcodes: single or double precision)
  - Three source fields, one destination field (4x5 = 20 bits)
  - Operation opcode: 5 bits
  - Set condition register flag .... Quiz

32 Pipelining

What is it?

Why does it speed up a process?

Minimum cycle time for a single pipeline stage

$$\tau = \max_i(\tau_i) + d = \tau_m + d \tag{1}$$

- τ_m is the maximum delay of combinational logic for any single stage.
- d is the time delay for a latch for signals to propagate from the input to the output.

Speedup

$$\text{Speedup} = \frac{\text{Time to execute N instructions on 1 stage}}{\text{Time to execute N instructions on K stages}} = \frac{N\tau_1}{[k + (N-1)]\tau_m} \tag{2}$$

Latch delays mean speedup will always be less than k. Increases in control logic may affect τ_m.

PIC Micro-controller: two stage pipeline

- Each stage consists of four clock cycles Q1-Q4
- Fetch increments the PC in Q1, loads the IR in Q4
- Execute uses all four cycles to move data, as necesary
- Any instruction that modifies the PC takes an extra macro-cycle

Modern pipeline (e.g. Pentium / PowerPC):

- Instruction fetch unit: get instructions
- Decode the instructions
- Identify where the operands are
- Fetch the operands to the right place
- Execute the operation
- Store the operands
- Operation Commit Unit: update the architectural state of the CPU

What are the issues?

- Branching / instruction fetches: pipeline flushes (UltraSparc: compiler had to insert NOPs)
- More stages means more control hardware: lower bound on cycle time
- Data dependencies: pipeline stalls
- Resource dependencies: pipeline stalls
- Interrupts: pipeline flushes
- Move the data v. move the state

33 Branch Prediction

Strategies for dealing with branching in pipelined architectures:

- NOPs
- Multiple fetch streams: feasible for short pipelines
- Prefetch branch target: fetch the branch target instruction, just in case
- Speculative execution: execute both branches, commit only one
- Loop buffer: special cache for the most recently fetched instructions, including future ones
- Branch target cache: cache for the target of a branch, e.g. K-6 holds 16 bytes for the last 16 branches
- Return address cache: speculatively fetch instructions, K-6 holds a 16-entry return address stack
- Branch prediction

Branch prediction:

- roughly 10% of instructions are unconditional branches, 10-20% are conditional branches
- Predict never taken: minimizes extra page accesses and loads
- Predict always taken: true more than 50% of the time (e.g. loops)
- Predict by opcode: works better than 75% of the time
- Take branches that go back, never take branches forward
- Taken/Not taken switch: store a bit in cache to remember the history of the instruction, use more bits
- Branch history table: address / history / target address
- Branch key: Use an N bit history, treat it as a key into a lookup table
- Compiler predictions: add bits to the instruction to let the compiler predict, unroll definite loops

Branch prediction on the K-6 (97% accuracy): ”two-level adaptive branch direction prediction algorithm”

- Branch History Table: 8192 entry table that holds instruction address and history
- BHT holds k-bits of history for that instruction (0.5% increase for each 2 additional bits from 6-12)
- BHT history bits are a key into a Global Pattern Table with 2^k entries
- GPT has N state bits, governed by a state machine (e.g. 1-bit last taken / 2-bits w/hysteresis)
- System dynamically learns the statistics for different branching patterns

34 Branch Prediction and Speculative Execution Vulnerabilities

Review two-level adaptive branch prediction

- BHT: 8192 entry table
- BHT: k-bits of history, up to 12 bits continues to improve performance
- State machine to control the GPT
- All examples of a specific history update the GPT

Branch prediction requires speculative execution in order to be useful

Spectre/Meldown/Rowhammer

- Rowhammer: takes advantage of a physical vulnerability
- Spectre/Meltdown: take advantage of speculative execution
  - Speculative execution has side-effects, notably cache lines can be modified
  - An unknown value can be used to cause a cache line to load
  - If the cache line is already there, the load will be fast
  - If the cache line is not already there, the load will be slower
  - An attacking program can control which cache lines are loaded

35 Superscalar Architecture: More Speed

Parallelism to speed up computation

- Pipelining keeps the concept of a serial processor, but creates parallel capabilities
- Vector processors: SIMD processing - MMX, SSE, specialized usage
- MIMD: multiple instruction, multiple data - more common
- Superscalar architectures do MIMD processing behind the scenes
- Compute like a parallel machine, act like a serial/scalar machine

Dependencies

- True data dependency
- Procedural dependency (depends on branch prediction)
- Resource conflicts: integer processor, FPU, branch unit
- Output dependency: Write after Write [WAW]
- Anti-dependency: Write after Read [WAR]

Register renaming gets rid of WAW and WAR dependencies: architectural state concept

Instruction issue and completion policies

- In order issue, in order completion
- In order issue, out-of-order completion
- Out of order issue, out-of-order completion
- Out of order issue, in-order completion

In-order completion makes it easier to define the state of the computer.

Go through the examples

How could you implement register renaming?

Scoreboard:

- Put an N bit counter on each register indicating number of times it is a source for current ops. No write operation can take place until all pending ops have read the register.
- Use a 1-bit counter on each register to indicate if it is being written by a pending op.

Use the counters to figure out if you have RAW, WAR, or WAW dependencies.

Examples

CPU:

- two fetch/decode stages
- Three functional units
- Two write-back pipelines
- Instructions fetched 2 at a time
- Instruction issue stalls with a resource conflict

Instructions:

- I1 requires 2 cycles in execute stage
- I3 and I4 use the same functional unit
- I5 depends on the output value of I4
- I5 and I6 use the same functional unit, different than I3 and I4

In-order Issue / In-order Completion

D1	D2	F1	F2	F3	W1	W2	Cycle
I1	I2						1
I3	I4	I1	I2				2
I3	I4	I1				I2	3
	I4			I3	I1	I2	4
I5	I6			I4	I3		5
	I6		I5			I4	6
				I6	I5		7
						I6	8

I3 can't be issues in time-step 3 because the write-back stage would be full.

In-order Issue / Out-of-order Completion

D1	D2	F1	F2	F3	W1	W2	Cycle
I1	I2						1
I3	I4	I1	I2				2
	I4	I1		I3	I2		3
I5	I6			I4	I1	I3	4
			I5		I4		5
	I6		I6		I5		6
					I6		7
							8

Out-of-order Issue / Out-of-order Completion

Out of order issue means that many more instructions need to be available at any given cycle. The decode stage is, therefore, not a pipeline but a buffer of instructions, any of which can be executed if the inputs are ready.

- The state of the computer can be difficult to define
- The CPU can look-ahead to instructions that don't have dependencies
- Instructions that have been executed can wait to be committed until resources are available

D1	D2	F1	F2	F3	W1	W2	Cycle
I1	I2						1
I3	I4	I1	I2	I3			2
I5	I6	I1	I6	I4	I2	I3	3
			I5		I1	I4	4
					I5	I6	5
							6
							7
							8

In-order Issue / Out-of-order Completion

In-order completion allows for precise definition of interrupts, traps, and faults.

- The instruction queue holds the pool of instructions that can be executed
- Instructions that have been completed will wait in a buffer until it's their turn to be committed
- Multiple instructions can be committed simultaneously

D1	D2	F1	F2	F3	W1	W2	Cycle
I1	I2						1
I3	I4	I1	I2	I3			2
I5	I6	I1	I6	I4			3
			I5		I1	I2	4
					I3	I4	5
					I5	I6	6
							7

36 Superscalar and K-6 Design

HW Review

Finish superscalar examples.

K-6

One of the first CISC on RISC designs, AMD's first competitive chip

- Heavily pipelined: 6 major stages, with substages
- Superscalar: up to 6 RISC ops per clock cycle
- Speculative execution with branch prediction: two-stage adaptive prediction
- Out of order execution, in-order completion
- Branch Target Cache: 16 bytes from up to 16 prior branches

I-Cache: 64K 2-way SA, 64 bytes/line, loaded in 32-byte increments

Instruction buffer: 16 byes per cycle from I-cache or the branch target cache

Decoder: processes up to two x86 instructions per cycle, converts to RISC op Quads

Scheduler: central part of the design

- Holds 24 RISC ops as six OpQuads
- Each OpQuad entry holds the state of the op, including the result (18 bytes/RISCOp)
- OpQuads move through the scheduler as a unit, and are committed as a unit
- Almost any individual RISC op can be executed at any time
- Branch ops are executed after reaching row 3 (no point in checking for conditions earlier)
- Load and Store ops are likely to be executed early
- Scheduler uses scan chains to select ops and identify operands

Overall execution pipeline

37 More Speed

How do we expand superscalar capabilities?

- more functional units
- larger instruction queues

Limitations of superscalar

- resources
- instructions to execute
- in-order completion
- inadequate use of functional units

How can we get more executable instructions on each clock cycle?

- hyperthreading
- compiler optimizations: pre-loading, loop unrolling

Historical figures: have people talk about their person, make a timeline

38 Final Class

Homework Review

Quiz