Building Your Own Machine
The Universal Machine (UM)

Introduction

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
COMP 40 Roadmap

- Ramp up your Programming Skills
  - Big programs that teach you abstraction, pointers, locality, machine representations of data
    - Intel Assembler Programming *The Bomb!*
  - Building Useful Applications in your Language
    - Building a Language Processor on your Emulator
  - Emulating your own hardware in software
Introducing the Universal Machine
Why work with the UM?

- AMD64 is complicated – UM has 14 instructions
- Easy to implement (HW or SW)
- Learn more about assemblers and linkers
- Learn to write assembler code
- You will write an emulator for this machine – emulation is a wonderful and important technique!
- The term UM traces to the work of Alan Turing

Detailed specification for the UM will come with your next assignment
UM Highlights

- 8 32 bit registers (no floating point)
- 14 RISC-style (simple) instructions
- Load/store architecture: all manipulation done in registers
- Segmented memory
  - Executing code lives in segment 0
  - Programs can create and map new zero-filled segments
  - Any segment can be cloned to become the new code segment (0)
- Simple byte-oriented I/O
- Does not have explicit:
  - Jump/branch
  - Subtract
  - Shift
UM Architectural Overview
14 UM Instructions

- Only 2 instruction formats
- Arithmetic and data
  - Add, multiply, divide (not subtract!)
  - Load value
  - Conditional move
- Logical
  - Bitwise nand (~and)
- Memory management
  - Map/unmap segment
  - Segmented load/store
  - Load program
- I/O
  - Input (one byte), Output (one byte)
- Misc
  - Halt
14 UM Instructions

- **Arithmetic and data**
  - **Add**, multiply, divide (*not subtract*)
  - Load value
  - Conditional move

- **Logical**
  - Bitwise nand (~and)

- **Memory management**
  - Map/unmap segment
  - Segmented load/store
  - Load program

- **I/O**
  - Input (one byte), Output (one byte)

- **Misc**
  - Halt

Add: \( r[A] := (r[B] + r[C]) \mod 2^{32} \)

Documented in the form of assignment statements (register transfer language – RTL)
14 UM Instructions

- **Arithmetic and data**
  - Add, multiply, divide *(not subtract!)*
  - Load value
  - Conditional move

- **Logical**
  - Bitwise nand (~and)

- **Memory management**
  - Map/unmap segment
  - Segmented load/store
  - Load program

- **I/O**
  - Input (one byte), Output (one byte)

- **Misc**
  - Halt

**Instruction Format**

```
In: | Out: | RA | RB | RC |
---|---|---|---|---|
0 | 32 | 16 |
```

**Add:** $r[A] := (r[B] + r[C]) \mod 2^{32}$
14 UM Instructions

- **Arithmetic and data**
  - Add, multiply, divide (*not subtract!*)
  - Load value
  - Conditional move

- **Logical**
  - Bitwise nand (~and)

- **Memory management**
  - Map/unmap segment
  - Segmented load/store
  - Load program

- **I/O**
  - Input (one byte), Output (one byte)

- **Misc**
  - Halt

**Instruction Format**

- 4 bits to select one of 14 operations

**Add:** $r[A] := (r[B] + r[C]) \mod 2^{32}$
14 UM Instructions

- **Arithmetic and data**
  - Add, multiply, divide *(not subtract!)*
  - Load value
  - Conditional move

- **Logical**
  - Bitwise nand (~and)

- **Memory management**
  - Map/unmap segment
  - Segmented load/store
  - Load program

- **I/O**
  - Input (one byte), Output (one byte)

- ** Misc**
  - Halt

Instruction Format:

```
OP  RA  RB  RC
```

3 bits to select one of 8 registers

Add: $r[A] := (r[B] + r[C]) \mod 2^{32}
What’s different from AMD64?

- Very few instructions – can we do real work?
- Memory is *word addressable*: Pointers to *words* not bytes
- Segmented memory model
- AMD/Intel has I/O, but we did not study it
- No floating point
Next time we will discuss in more detail

- Memory models and the segmented memory of the UM
- I/O
- How programs start and execute
- Building complex instructions from simple ones
Computability
and
Turing-Completeness
Emulators
Emulators

- Hardware or software that implements another machine architecture

- Great way to learn about machines:
  - The emulator you write will do in software the same thing machines do in hw

- Terrific tool for
  - Testing code before hardware is ready
  - Performance analysis (e.g. our testcachesim)
  - Evaluating processor/memory design before committing to build chip
  - Moving systems to a new architecture (e.g. Apple move from PowerPC to Intel)
Question?

- How would you implement an emulator for the UM?