The Universal Machine (UM)

*Implementing the UM*

Noah Mendelsohn
Tufts University
Email: noah@cs.tufts.edu
Web: http://www.cs.tufts.edu/~noah
Review:
Overview of the UM
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
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
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**

```
32 16  0  
```

```
OP    R  R  R
      A  B  C
```

**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

32 bits instruction format with 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**

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

---

3 bits to select one of 8 registers
What’s different from AMD64?

- Very few instructions – can we do real work?
- Pointers to *words* not bytes
- Segmented memory model
- AMD/Intel has I/O, but we did not study it
- No floating point
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)
Very simplified view of computer
Very simplified view of UM
Very simplified view of UM
Instructions fetched and decoded

Add instruction

Nand (not and) instruction

Memory

3 4 6 2
6 2 5 1
Instructions fetched and decoded

ALU

Arithmetic and Logic Unit executes instructions like add and shift updating registers.
Questions?

- How would you implement an emulator for the UM?

Specific questions:
- What state must your emulator program keep and update?
- What logic must it perform?
- What external interfaces does it have?
- What is the initial state of the machine and what must be available in the environment (e.g. where does the program come from?)

Remember: you’ll have two programs running at once!
1. Your emulator is a C program
2. It is simulating the execution of a UM program
Segmented Memory
Why segmented memory?

- **Advantages**
  - Allows programs with small pointers to address lots of memory
  - Illustrates a model that was popular when circuits were limited
  - For class: gives you an interesting programming challenge 😊

- **Disadvantages**
  - Simple is better: segmented memories are more complex to use
  - Few modern machines and operating systems use segmented architectures
Traditional memory

```assembly
mov (%rbp), %rax
```

Diagram:
- `rbp`
- `Some address`
- `rax`
- `Value`
- `0`
- `Last addr`
- Memory block with `Value`
Segmented memory in UM

\[ r_7 := m[r_1][r_5] \]
Segmented memory in UM

```c
r7 := m[r1][r5]
```

Diagram: 
- `r7 := m[r1][r5]`
- Identify segment
- Segment 0: 0-4
- Segment 1: 4-8
- Segment 2: 8-12
- Segment 3: 12-16

Value: 4
Segmented memory in UM

\[ r7 := m[r1][r5] \]
Thinking about segmented memory

A Segmented Memory Acts Like A Machine with Multiple Memories

To address data you need a [segid, offset] pair

\[
\text{r7 := m[r1][r5]} \quad \text{(Identify segment)}
\]

\[
\text{r1} \quad \text{2 (Address in segment)}
\]

\[
\text{r5} \quad \text{4 (Address in segment)}
\]

\[
\text{r7} \quad \text{Value (Value)}
\]
Your program controls segment creation
Your program controls segment creation

Map segment instruction

$r[C]$ gives length for new segment

Segment 0

Segment 1

Segment 2

Segment 3
Your program controls segment creation

System puts ID of new segment in $r[B]$

Map segment instruction

$r[C]$ gives length for new segment

Segment 0

Segment 1

Segment 2

Segment 3

(new!)Segment 4

Length set from $r[C]$
Where do these segments really live?

Segment 0

Segment 2

Segment 1
Where do these segments really live?

In real machines, the hardware and operating system together keep track of which segment is where...hardware usually has registers that can hold start address and length of segments the program is using.
How Segments are Used in the UM
How segments are used in the UM

- **Running code is always in segment zero**
  - The emulator arranges for initial code image to be in segment zero
  - Copies of other segments can be swapped in to replace segment zero, thus changing what code is running

- **Programs can create (map), use, and release (unmap) other segments to hold data, etc.**

- **The emulator keeps track of:**
  - Which segment ids map to what actual data
  - Which segment ids are available as names for new segments
  - *Keeping track of all this is a significant part of the work you will do in your emulator*
  - *You can malloc space for new segments when needed*
Summary
What we learned today

- Review the UM: simple 14 instruction RISC machine
- **Emulators: an important and powerful concept**
  - Testing code before machines are ready
  - Proving machine designs work
  - Performance evaluation of machines and programs
  - Great way to learn about machines
- **To build an emulator:**
  - Consider: state, logic, and external interfaces
  - Emulators often match the structure of real hardware implementations
- **Segmented memories**
  - Used when main memory is too large to address with pointers in registers
  - Segment ids select what appear to be multiple smaller memories, each addressed from zero
  - OS and hardware work together to map segments to flat underlying memory