Universal Machine

For this unit, you'll write an *emulator* (contrast with *simulator*) for a simple computer at the machine architecture/ISA level.

A Turing-complete machine in 14 instructions

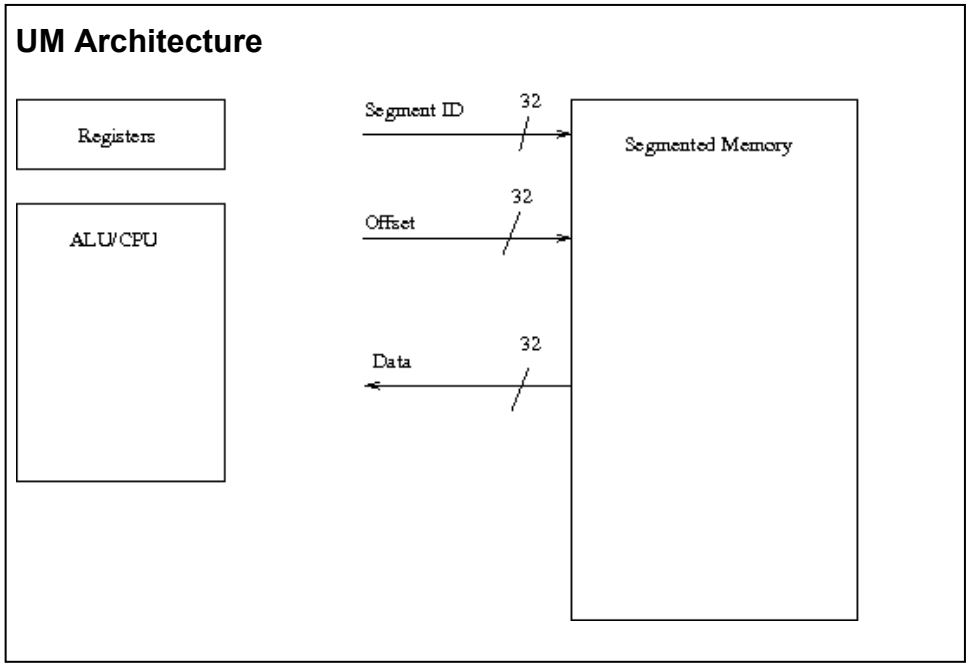
- **Caveat:** The UM is Turing-complete only in the sense that the Intel box is Turing-complete: we can simulate any computation until we run out of memory. And yes, you can write a UM interpreter in UM assembly language. This makes it "universal".

> Aside: What is the smallest instruction set a machine can have and be universal?

Why the Universal machine?

- Learn to distinguish binary machine code from assembly code
- Learn how a sensible binary instruction set works
- Possibly learn what the assembler and linker do
- Possibly write a little assembly code
- Finally, **processor emulation** interesting in its own right (distant cousin to Java Virtual Machine)

Architecture of the Universal Machine



A 32-bit machine:

- 32-bit registers and 32-bit addresses

A **segmented memory architecture**:

- Most resembles the Intel 8086 (first IBM PC)
- Access to any memory element requires *two* registers:
  - "segment" register
  - offset within the segment

  Segmentation was used on the first IBM PC (foundation of Microsoft's success), but it is now out of fashion

Word-oriented, RISC instruction set architecture

- No byte instructions; smallest addressable unit is the word (Like CRAY-1 supercomputer)
- All computation in registers, pure load/store (RISC)
- Every instruction is 32 bits wide (RISC) --- can find "next instruction" without *decoding* current instruction
- Only two binary machine-code formats (RISC) --- very fast instruction decoding

Computational capabilities deliberately limited:

- Simpler to implement and debug
- Limitations can be worked around by a compiler or Macro Assembler
- Purpose is partly instructional, e.g., learn how to implement subtraction in terms of other operations
- No floating-point operations

Instruction set

- Computation: Add, multiply, divide, NAND
- Data movement: load and store from segment, load value
- Control flow: computed goto ("run instruction in segment")
- Decision-making: conditional move
- Create and destroy segment (activate and inactivate)
- Input and output of 8-bit characters
- Halt

Emulating the Universal Machine

Keep representation of machine state:

- Contents of 8 registers

- Contents and sizes of 1 or more segments; segment 0 is always the program
- Value of the program counter

Interface: given an initial 'segment 0' (binary program), run until reaching a `halt` instruction, then return.

Design of UM software

Asking for a more modular design:

- UM execution should be a standalone component with its own API
- UM loading (from a file) should be a standalone component
- `main()` should be almost empty — just stitch together
- The biggest design task is emulating segments

UM Design and Documentation

Reminder: Instruction set

- Computation: Add, multiply, divide, NAND
- Data movement: load and store from segment, load value
- Control flow: computed goto ("run instruction in segment")
- Decision-making: conditional move
- Create and destroy segment (activate and inactivate)
- Input and output
- Halt

See Figure 1 in the assignment for a kind of RTL semantics of the instructions.

Opcode	Instr	Semantics
0x00	CMOV	if `$r[C]` ≠ `0` then `$r[A] := $r[B]`
0x01	SLOAD	`$r[A] := $m[$r[B]][$r[C]]`
0x02	SSTORE	`$m[$r[A]][$r[B]] := $r[C]`
0x03	ADD	`$r[A] := ($r[B] + $r[C]) mod` 2^{32}
0x04	MULT	
0x05	DIV	
0x06	NAND	`$r[A] := ¬($r[B] ∧ $r[C])`
0x07	HALT	

0x08	MAP	
0x09	UNMAP	
0x0A	OUTPUT	
0x0B	INPUT	
0x0C	LOADP	
0x0D	LOADV	

Emulating the Universal Machine

Keep representation of machine state:

- Contents of 8 registers
- Contents and sizes of 1 or more segments; segment 0 is always the program
- Value of the program counter

Interface: given an initial 'segment 0' (binary program), run until reaching a `halt` instruction, then return.

Computational model: fetch-decode-execute.

Design of UM software

Asking for a more modular design:

- UM execution should be a standalone component with its own API
- UM loading (from a file) should be a standalone component
- `main()` should be almost empty — just stitch together
- The biggest design task is simulating segments

Representation is still the essence of programming.

Representation

Let's look at the data in the problem:

- Universal Machine word (`uint32_t` or `int32_t`)
- Universal Machine segment (sequence of words of *fixed size*)

  Up to *three* representations:

  - External, on disk (fixed by problem)
  - Internal, can grow one word at a time (recommended by instructor)
  - Internal, used during emulation (chosen by you)

  Questions:

    - Which pairs of representations can be the same?
    - Which modules should know about which representations?
- Set of segment IDs partitioned into **mapped** and **unmapped** subsets
- Segment mapping (from mapped IDs to segments)
- Universal Machine general-purpose registers
- Universal Machine program counter
- Universal Machine instructions
    - Representation within a segment is fixed
    - Representation used in emulator can be chosen freely

What invariants are true even in the world of ideas (about the *abstractions*)?

Which modules should know about which *representations*?

Which representations should be **public**?

Which representations should be **private**?

UM segmented memory architecture

Modeled on design of original x86 architecture. (Recall: x64 has 2^{64} addresses, but can only use 2^{48}.) Original PC was 16-bit. Rather than have 2^{16} locations, have 2^{16} location segments — a segment is an address space (think *namespace*).

World of ideas:

- Q: An *active* 32-bit identifier is associated with what?
    - A: Sequence of up to 2^{32} mutable 32-bit locations
- We have a bijection between *active* identifiers and sequences (up to 2^{32} active segments).
- Q: How is the bijection used and changed by the instruction set?
    - A: Examine each instruction
- Q: When an identifier makes the transition from active to inactive, what happens to its sequence?

World of implementation

- How is a 32-bit identifier represented?
- How can we distinguish active from inactive identifiers?
- How is a sequence of mutable 32-bit locations represented?

Do we want to limit memory? Yes, but not internally. For testing, debugging, grading, run with memory and CPU limits (see assignment for how to do this).

Documentation (as expected for UM)

UM Documentation

Three layers:

- Architecture
- Interfaces
- Implementations

Consider your audience. Don't leave old (commented out) code lying around.

Here is what I expect for documentation of the Universal Machine:

- A short README that:
    - States briefly that the directory contains a UM emulator, with simple instructions for how to build and use it.
    - Identifies you and your programming partner by name
    - Acknowledges help you may have received from or collaborative work you may have undertaken with others
    - Explains how long it takes your UM to execute 50 million instructions, and how you know
    - Mentions each UM unit test (from UMTESTS) by name, explaining what each one tests and how
    - Says approximately how many hours you have spent analyzing the assignment
    - Says approximately how many hours you have spent preparing your design
    - *Briefly discussess performance achieved with gcc -O1 and -O2 optimization levels
    - *Says approximately how many hours you have spent solving the problems after your analysis
    - *Any significant departures from your design
    - *Identifies what has been correctly implemented and what has not

  The items with (*) are not likely to be appropriate in your initial design submission; add them for your final submission. Insofar as things change between design and final submission, update any of the items in your README.

- In your DESIGN or design.pdf:
    - Documentation of the *architecture* of the system.
        - Each *interface* may get a short description, but no more.
        - Rule of thumb: data types may be mentioned (in the world of ideas) but functions should not be mentioned.
    - Each *interface* to be documented with
        - Overall header comment *briefly* explaining

        - The general purpose (in many cases one line is enough)
        - The declared data types and what they stand for in the world of ideas
        - General knowledge needed to understand and use the interface (e.g., references to man pages or other info), if any
      - **Well-chosen names!!** (critical form of documentation)
      - Short documentation for each function. Aim for about one line per function — but recognize that sometimes it is clearer to document a small group of functions and let the names and types speak for themselves.

        Don't forget to **document the element types of arrays**!!
    - Each *implementation* to be documented with
      - Descriptions for each declared type and global variable. What does it represent in the world of ideas? What are its *(representation) invariants*?
      - Descriptions of any functions private to the implementation:
        - These functions should have prototypes and be declared `static`
        - The prototypes should be documented in much the same way as the function prototypes in a .h file
      - Bodies of functions should be documented only where code is tricky, long, or difficult to understand. Most of you are getting this part right already.
- Much simpler modular structure than image compression
    - Existing modules including `Bitpack`, `Seq`, `Array`, `Table`, and so on. Ample choices available.
      - Yes, you can use your Bitpack if you wish!
      - Yes, we will give you a debugged bitpack if you prefer that.
    - A module to manage UM memory access
    - A module to run UM codes (execution phase)
    - A module to isolate the secret of the on-disk format
    - Some main functions

What I'm looking for in your code and documentation

Documentation:

- *Stratify* into three layers: architecture, interfaces, implementations.

- Internal functions should be `static` and be documented like interface functions, but *not* `inline` without good reason. (Usually the compiler makes good inlining decisions on static functions.)

- Don't forget to try compiling with *both* `-O1` and `-O2` to see about your performance target — mention results in README

Code:

- You have a lot of freedom to change design decisions within modules.

- An important aspect of modular reasoning is to identify *design decisions that are likely to change*. Many of you have already done so, especially around UM segments. I want those decisions **hidden behind a layer of abstraction**.

- Otherwise, I'm mainly looking for good names:

    - There are a lot of numbers in binary code. The number 3 means something different depending on whether it occurs in opcode position, in register position, or in value position (in the Load Value) instruction. I expect you to use distinct names in your code. Where you need to, **define enumeration types**.

      (Remember the example of the seven image transformations.)

    - Make sure your naming conventions distinguish the *name* of a container (register 3) from the *contents* of that container (the integer 17, a 32-bit segment identifier).

    - If you choose to define auxiliary functions, remember modular reasoning, and be sure that you separate these two concerns:

        - Instruction decoding

        - How the segmented memory unit works

      For example, if you have an auxiliary function that finds a memory location, its parameters should be called *segment register* and *offset register*, not `rB` and `rC`. In fact, the segmented load and store instructions don't even use the same registers!

More on Segments

Q: Can one use a Hanson Sequence? Yes, but … Sequences are bounds checked and re-sizable. Segments aren't re-sizable (though they are reassignable), and bounds checking is not necessary. Consider overhead and what it buys you.

Reminder: Multiple formats of segments.

- On disk
- Executing

Could use sequence for reading a file. Can also seek to end and allocate array. Could use dynamic arrays/vectors.

Mapping a segment does *not* take a segment ID — like `malloc()`, just takes size.

For unit tests, you need to know what you're testing! If you have a test named `test1`, that tells me nothing. `test_length` is more suggestive.

What does test prove? Avoid redundant tests. What faults does test uncover? 2 sets of tests:

- UM binary: submit with assignment. Used on other people's code. Discussed in design.
- Segment interface tests: submit with design. Students designed interface, so no universal tests. But you need this to work!

You'll appreciate unit tests in the next assignment when you profile and optimize code (optimize without changing interfaces).

Noah Mendelsohn (noah@cs.tufts.edu)
Last Modified 10 March 2014