Midterm Review

CSE 410

Lecture 10
Midterm Practical Matters

• The midterm will be available as a Canvas quiz
• It will be available during class time (11:30-12:20) on Monday, 2/7/22
• You can take it from anywhere
  • If you’re on campus and want to take it in the classroom, you’re welcome to do so
• You can use any resource that you can’t talk to

• Please do not post questions on the discussion board about the midterm (until late Monday night)
• The course staff will be answering questions by:
  • looking for email sent to cse410-staff@cs.Washington.edu
  • I will also be in the classroom on campus and can answer questions in person
Midterm Material and Resources

• The material is the course material up to but not including the datapath (so, Lectures 1-7)

• Study materials are the homeworks, the posted homework solutions, the slides, and the lectures

• We will attempt to answer questions emailed to cse410-staff@cs.Washington.edu as promptly as we can over the weekend

• There will be extra office hours over the weekend
  • E.g., I have one set up for 3:00-4:00 on Sunday
What This Course is About: Interfaces and Representations
Theme: Interfaces and Layering

No Layering

Layering & Translation
RISC-V ISA: Load-Store Architecture

Example Instruction:
“Add register 3 to register 4 and put the result in register 3”
ISA Key Ideas: Values / Variables / Memory

• Memory: big, slow
• Registers: limited number, fast
• CPU operates only on values in registers
  • load/store are only memory operations
  • simple(r is faster)
• In general, variables have a “long term” location for their values in memory but will have their values in registers while being actively used
• One of the compiler’s jobs is to make good use of registers
  • minimize the number of loads and stores required to perform the computation
• Base-displacement addressing
  • Why?
    • Example: arrays
    • Example: local variables
ISA Key Ideas: Instructions

• Every instruction is 32 bits
  • `simple(r is faster)`
  • limits the number of bits available to specify parts of the instruction
    • How big can an immediate value be?
      • Why are there instructions that use immediate values?
    • E.g., how might you change the instruction encoding so that the CPU could have 63 registers instead of 32?

• Instructions (can) modify state
  • The value in a register
    • `add x3, x2, x1`
  • The value of the program counter
    • `bne x2, x3, loop`
  • A value in memory
    • `sw x5, -24(sp)`

• Processors don’t execute programs, they execute one instruction after another
  • “Programs” are an abstraction created by higher layers
  • “State machine” → fast
Assembly Language and Assemblers

• The ISA defines what memory resources exist and what instructions exist
• It defines a representation of instructions as bit strings
• Bit strings are handy for the CPU to decode, but not for anyone else

• Assembly language is just a more readable version of machine code, along with a tiny bit of very straightforward processing
  • labels let the programmer talk about a location without having to compute the offset the machine instruction requires
  • the assembler can easily compute the offsets when given a label

• There are no higher level constructs, though
  • No procedures
  • No local variables

• Those are created by the way in which the ISA resources and instructions are used
  • Layering
Layering Languages above the ISA

• The ISA supports only very simple operations
  • Simpler is faster

• It’s tedious and error prone to express computations in the ISA
  • Assembler is just a more human readable representation of the instructions the hardware can actually execute
    • Roughly like “ten” versus “10”

• Compilers are translators from one interface (the language definition) to another (e.g., instructions in the ISA)

• The higher level language has, roughly, three things:
  • variables (values, memory)
  • expressions, like \( X + Y*Z \) (load and arithmetic instructions)
  • control flow like loops and subroutines (branches and jump-and-link)
  • (what about types?)
Compilers

• By layering a higher level language on top of the ISA, we get
  • More powerful statements for the programmer to use, which makes programming simpler and less error prone
  • A simple ISA that can be implemented in a way that is very fast
  • Automatic and error free translation from the language interface to the ISA interface

• Compilers do their work *statically*

• The semantics of the higher level language can be made even more powerful (in some cases) by deferring some of the compiler’s job to run time (doing it dynamically)
  • E.g., dynamically typed languages
  • E.g., run time libraries
Binary Representation

• At run time, everything is represented as bits
  • instructions
  • numbers
  • strings
  • true / false
  • arrays
  • objects

• Why binary?

• Numbers can be written in many ways, e.g., decimal, binary, hexadecimal
• Hex is handy because (a) it’s relatively short, and (b) each hex digit represents a string of 4 bits. (A decimal digit represents a string of 3.32 bits...)
Binary Integers

<table>
<thead>
<tr>
<th></th>
<th>000</th>
<th>001</th>
<th>010</th>
<th>011</th>
<th>100</th>
<th>101</th>
<th>110</th>
<th>111</th>
</tr>
</thead>
<tbody>
<tr>
<td>signed</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>-4</td>
<td>-3</td>
<td>-2</td>
<td>-1</td>
</tr>
<tr>
<td>unsigned</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

• Two’s complement representation: why?
  • Need a way to represent negative numbers
  • Could use a bit to indicate negative or non-negative, but then binary arithmetic is clumsy
  • We can add bit strings representing two’s complement signed integers just like we add unsigned integers
    • simpler is faster

• Why have unsigned numbers?
Floating Point

• 6.0221409 x 10^{23} in decimal

• How would we represent it in a fixed number of bits
  • Use some bits to represent the exponent (here 23)
  • Use some bits to represent the mantissa (her 6.0221409)
  • Don’t need any bits to represent the base (here 10) because it’s always the same (defined by the standard, typically 2)
### Instruction encoding / classes

<table>
<thead>
<tr>
<th>CORE INSTRUCTION FORMATS</th>
</tr>
</thead>
<tbody>
<tr>
<td>31</td>
</tr>
<tr>
<td>R</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td>U</td>
</tr>
</tbody>
</table>

- **R type**
  - add, sub, sll, slt, sltu, xor, srl, sra, or, and,

- **I type**
  - lw, lb, addi, slli, slti, sltiu, xori, srli, srai, ori, andi

- **SB type**
  - beq, bne, blt, bge, bltu, bgeu
PC-Relative Branching

• Idea: use the PC as the implicit base register
  • Target address = PC + offset
  • Don’t have to specify a base register in the instruction encoding (because the PC is always the base register)
  • That gives you full 13 bits to hold the offset
  • Might want branch forward in instruction stream, or might want to branch back
    • Make the offset a signed value
      • -4096 to 4095

\[
bne \ x1, x2, offset
\]
Arrays

- Arrays are contiguous block of memory that we think of as composed of a number of pieces of identical size.
- “Contiguous” and “identical size” allow us to translate the array concept of indexing, e.g., A[3], into a simple calculation:
  - A[3] is at base address of A plus 3 times the size of each element.
  - If x6 holds the address of the first byte of A, then A[3] is at 12(x6).

- If the compiler translates A[3] into 12(x6), there is no array bounds checking.
- IF the language wants to do array bound checking, the compiler must generate instructions to check the index (unless it can figure it out statically):
  - int A[100];
  - int i = A[101]; // error? when?
Objects (Structures)

• Objects are contiguous blocks of memory holding elements that may be of different sizes

• The compiler determines statically:
  • how big each object is
  • what the offset is for each element within the object

• For example,
  • class Person {
      int id;
      int phoneNumber;
  }
  • A Person object is 8 bytes long.
  • If x6 is the base address of a Person, 0(x6) is where that Person’s id is stored and 4(x6) is where phoneNumber is stored
    • Or, could be id at 4(x6) and phone at 0(x6) – it doesn’t matter
Compiling a C Program

```c
int val = 10238;
int i;
int main(int argc, char *argv[]) {
    int i;
    for (i = 2; i <= val/2; i++) {
        if (((val/i)*i == val)) {
            printf("%d\n", i);
        }
    }
    return 0;
}
```

```assembly
lw  x8, i
beq  x0, x0, test
body:
  <body code>
addi x8, x8, 1
sw  x8, i
test: lw  x9, val
srai x9, x9, 1
blt  x8, x9, body
beq  x8, x9, body
```
The Memory Model

- While compiling the code, the compiler “knows” what memory will look like at run time.
- The OS (program loader) knows the same thing.

```
Subroutine
args & locals

"new" (malloc)

.data section

.text section

Stack

Heap

Static Data

Instructions
```
int sub(int w) {
    int x, y, z;
    ...
    return x;
}

val = sub(2);
This class’s `getArea()` instructions in memory

Shape larger(Shape A, Shape B) {
    if ( A.getArea() > B.getArea() )
        return A;
    return B;
}

What problem does this solve?
Boolean Logic / Gates

- Digital circuits are built out of digital gates
- Each gate implements some logic function
Example Boolean Circuit

\( \neg A \land B \lor (A \land \neg B) \)

Exclusive Or

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
Full (One Bit) Adder

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

A diagram of a full (one bit) adder is also shown.
4-bit full (ripple carry) adder
Sequential Circuit: Operation

- At time \( n \) the memory elements have some values
  - The combinational circuit has “settled” and its output are stable (unchanging)
  - If we update the memory elements values, though, the outputs of the combinational circuit change
Implementing sequential components: the gated d-latch

- Component stores 1 bit, and advertises both it’s value ($Q$) and the negation of its value ($\overline{Q}$)
- When $C$(lock)=1 the output Q records the value of D
  - if $D=1$ then $R=0$ and $S=1$. $R=0$ makes $Q=1$. $Q=1$ makes $\overline{Q}=0$.
  - if $D=0$ then $R=1$ and $S=0$. $S=0$ makes $\overline{Q}=1$, which makes $Q=0$.
- When $C=0$ the output ignores the value of D
  - both $R$ and $S$ are 1. If $Q=1$ then $\overline{Q}$ is 0 – no change. If $Q=0$, then $\overline{Q}$ is 1 – no change.
Synthesize a Boolean Circuit (Multiplexor)

<table>
<thead>
<tr>
<th>s</th>
<th>i0</th>
<th>i1</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

\((\neg s \land i_0) \lor (s \land i_1)\)
Summary

- Make sure the assignments (and their solutions) make sense to you
- Look at the slides
- Re-watch lectures as needed
- Ask questions by email or in office hours