Module 11: Models of Computation.
Final Exam information

- Announcements:
  - Final exam: Saturday December 12\textsuperscript{th}, 8:30.
  - Office Hours:
    - In the week before the exam.
    - The schedule will be posted on piazza.
Final Exam Information

- Final exam details:
  - 2h 20 minutes + upload time
  - Covers everything discussed in the course
    - includes labs, although you don't need to memorize all of the solutions.
  - Slightly more emphasis on later topics.
    - Approximately 50 minutes take home test #3 plus 90 minutes evenly distributed + 15 minutes upload.
  - You can use anything that is available on the course web site, canvas or in your textbook.
  - No other help allowed.
Module 11: Models of Computation

- By the start of class, you should be able to:
  - Define the terms domain, co-domain, range, image, and pre-image
  - Use appropriate function syntax to relate these terms (e.g., \( f : A \rightarrow B \) indicates that \( f \) is a function mapping domain \( A \) to co-domain \( B \)).
  - Determine whether \( f : A \rightarrow B \) is a function given a definition for \( f \) as an equation or arrow diagram.
Module 11: Models of Computation.

- CPSC 121: the BIG questions:
  1. How can we build a computer that is able to execute a user-defined program?

- We are finally able to answer this question.
  - Our answer builds up on many of the topics you learned about in the labs since the beginning of the term.

- More generally:
  - What can we compute?
  - Are there problems we can not solve?
Module 11: Models of Computation

- Module Summary
  - (a bit of) Computing history.
  - A working computer.
  - DFAs and regular expressions.
  - Computations that we are unable to perform.
  - Appendix: working computer details.
Module 11.2: (a bit of) Computing history

- Historical notes:
  - Early 19th century:
    - Joseph Marie Charles dit Jacquard used punched paper cards to program looms.
Module 11.2: (a bit of) Computing history

- Historical notes (early 19th century continued):
  - Charles Babbage designed (1837) but could not build the first programmable (mechanical) computer, based on Jacquard's idea.

http://www.computerhistory.org/babbage/
Module 11.2: (a bit of) Computing history

- Historical notes (continued):
  - 1941: Konrad Zuse builds the first electromechanical computer.
    - It had binary arithmetic, including floating point.
    - It was programmable.
  - 1946: the ENIAC was the first programmable electronic computer.
    - It used decimal arithmetic.
    - Reprogramming meant rewiring.
    - All its programmers were women.
Module 11.2: (a bit of) Computing history

- Historical notes (mid 20\textsuperscript{th} century, continued)
  - The first stored-program electronic computers were developed from 1945 to 1950.
  - Programs and data were stored on punched cards.
Module 11.2: (a bit of) Computing history

- A quick roadmap through our courses:
  - **CPSC 121**: learn about gates, and how we can use them to design a circuit that executes very simple instructions.
  - **CPSC 213**: learn how the constructs available in languages such as Racket, C, C++ or Java are implemented using these simple instructions.
  - **CPSC 313**: learn how we can design computers that execute programs efficiently and meet the needs of modern operating systems.
Module 11: Models of Computation

- Module Summary
  - (a bit of) Computing history.
  - A working computer.
  - DFAs and regular expressions.
  - Computations that we are unable to perform.
  - Appendix: working computer details.
Module 11.3: A working computer

- **Von-Neumann architecture**

![Diagram showing the components of a computer system: Memory, Control Unit, Arithmetic & Logic Unit, and Input/Output. Memory contains both programs and data. The CPU (Central Processing Unit) is connected to Memory, Control Unit, and Arithmetic & Logic Unit. The Input/Output is also connected to the CPU.](image-url)
Module 11.3: A working computer

- Memory
  - Contains both instructions and data.
  - Divided into a number of memory locations
    - Think of positions in a list: \( \text{list-ref mylist pos} \)
    - Or in an array: \text{myarray[pos]} \ or \text{arrayList: arrayl.get(pos)}.
Module 11.3: A working computer

- **Memory**
  - Each memory location contains a fixed number of bits.
    - Most commonly this number is 8.
    - Values that use more than 8 bits are stored in multiple consecutive memory locations.
      - Characters use 8 bits (ASCII) or 16/32 (Unicode).
      - Integers use 32 or 64 bits.
      - Floating point numbers use 32, 64 or 80 bits.
Module 11.3: A working computer

- Arithmetic and Logic Unit
  - Performs arithmetic and logical operations (+, -, *, /, and, or, etc).

- Control Unit
  - Decides which instructions to execute.
  - Executes these instructions sequentially.
    - Not quite true, but this is how it appears to the user.
Module 11.3: A working computer

- Our working computer:
  - Implements the design presented in the textbook by Bryant and O'Hallaron (used for CPSC 213/313).
  - A small subset of the IA32 (Intel 32-bit) architecture.
  - It has
    - 12 types of instructions.
    - One program counter register (PC)
      - contains the address of the next instruction.
    - 8 general-purpose 32-bit registers
      - each of them contains one 32 bit value.
  - used for values that we are currently working with.
Module 11.3: A working computer

- Example instruction 1: `irmovl 0x1A, %ecx`
  - This instruction stores a constant in a register.
  - In this case, the value 1A (hexadecimal) is stored in `%ecx`.

- Example instruction 2: `subl %eax, %ebx`
  - The `subl` instruction subtracts its arguments.
  - The names `%eax` and `%ebx` refer to two registers.
  - This instruction takes the value contained in `%eax`, subtracts it from the value contained in `%ebx`, and stores the result back in `%ebx`. 
Module 10.2: Implementing a working computer

- Example instruction 3: \texttt{jge \$1000}
  - This is a \texttt{conditional jump} instruction.
  - It checks to see if the result of the last arithmetic or logic operation was zero or positive (Greater than or Equal to 0).
  - If so, the next instruction is the instruction stored in memory address \texttt{1000} (hexadecimal).
  - If not, the next instruction is the instruction that follows the \texttt{jge} instruction.
Module 11.3: A working computer

- Sample program:
  
  `irmovl 0x3,%eax`
  `irmovl 0x35, %ebx`
  `subl %eax, %ebx`
  `halt`
Module 11.3: A working computer

How does the computer know which instruction does what?

- Each instruction is a sequence of 8 to 48 bits.
- Some of the bits tell it which instruction it is.
- Other bits tell it what operands to use.

- These bits are used as select inputs for several multiplexers.
Module 11.3: A working computer

- Example:
Module 10.2: Implementing a working computer

- Example 1: `subl %eax, %ebx`
  - Represented by `6103` (hexadecimal)
  - arithmetic or logic operation (the use of “6” to represent them instead of 0 or F or any other value is completely arbitrary).
Module 11.3: A working computer

- Example 2: `irmaovl 0xfacade, %ecx`
  - Represented by
    
    $\text{30F100FACADE (hexadecimal)}$
    
    $\text{$0xfacade}$
    
    $\text{%ecx}$
    
    no register here
    
    ignored
    
    move constant into a register
Module 11.3: A working computer

- How is an instruction executed?
  - This CPU divides the execution into 6 stages:
    - Fetch: read instruction and decide on new PC value
    - Decode: read values from registers
    - Execute: use the ALU to perform computations
    - Memory: read data from or write data to memory
    - Write-back: store value(s) into register(s).
    - PC update: store the new PC value.
  - Not all stages do something for every instruction.
Module 10.2: Implementing a working computer

- Example 1: `irmovl 0xfacade, %ecx`
  - Fetch: current instruction ← 30F100FACADE
    next PC value ← current PC value + 6
  - Decode: nothing needs to be done
  - Execute: valE ← valC
  - Memory: nothing needs to be done
  - Write-back: R[%ecx] ← valE
  - PC update: PC ← next PC value
Module 10.2: Implementing a working computer

- Example 2: `subl %eax, %ebx`
  - Fetch: current instruction ← 6103
    - next PC value ← current PC value + 2
  - Decode: valA ← value of %eax
    - valB ← value of %ebx
  - Execute: valE ← valB - valA
  - Memory: nothing needs to be done.
  - Write-back: %ebx ← valE
  - PC update: PC ← next PC value
Module 11: Models of Computation

- Module Summary
  - (a bit of) Computing history.
  - A working computer.
  - **DFAs and regular expressions.**
  - Computations that we are unable to perform.
  - Appendix: working computer details.
Module 11.1: DFAs and regular expressions

- Sets and functions can be used to define many useful structures.
- Example: we can definite valid DFAs formally: a DFA is a 5-tuple \((\Sigma, S, s_0, F, \delta)\) where
  - \(\Sigma\) is a finite set of characters (input alphabet).
  - \(S\) is a finite set of states.
  - \(s_0 \in S\) is the initial state.
  - \(F \subseteq S\) is the set of accepting states.
  - \(\delta: S \times \Sigma \rightarrow S\) is the transition function.
Module 11.1: DFAs and regular expressions

- DFAs and Regular expressions.
  - In lab 7, you wrote regular expressions matching the patterns we gave you.
  - Regular expressions are very useful when you need to read and validate input.
  - Many modern programming languages provide functions that allow you to manipulate regular expressions:
    - Dr. Racket, Java, Python, Perl, etc.
Module 11.1: DFAs and regular expressions

- How does a program determine if an input string matches a regular expression?
  - **Theorem**: every set of strings matched by a regular expression can be recognized by a DFA.
  - Hence the functions provided by the language build a DFA corresponding to the regular expression.
  - And then this DFA is given the input string one character at a time.
Module 11.1: DFAs and regular expressions

- This result goes both ways:
  - if a set of strings is accepted by a DFA, then there is a regular expression for this set of strings.
  - In this sense, DFAs are exactly as “powerful” as regular expressions (no more, no less).

- How do we build the DFA given a regular expression?
  - First we build a NFA (non-deterministic finite-state automaton) for the regular expression.
  - Then we convert the NFA into a DFA.
Module 11.1: DFAs and regular expressions

• What is a NFA?
  • It is like a DFA but
    ▷ There can be multiple arrows with the same label leaving from a state
    ▷ There can be arrows labelled $\varepsilon$ that we can take without reading the next input character.
    ▷ So we can sometimes choose which state to go to.
  • A NFA accepts a string if at least one sequence of choices leads to an accepting state.
Module 11.1: DFAs and regular expressions

- Example:
Module 11.1: DFAs and regular expressions

- What regular expression corresponds to the strings that this NFA accepts?

  a) $\varepsilon \mid ab$
  b) $\varepsilon \mid abaa$
  c) $\varepsilon \mid ab \mid abaa$
  d) $\varepsilon \mid ab \mid (aba)^+a$
  e) None of the above.
Module 11.1: DFAs and regular expressions

- **Theorem**: we can transform every regular expression into a NFA with
  - Exactly one accepting state
  - No arcs pointing to the initial state
  - No arcs leaving the accepting state

- **Fact**: every regular expression can be rewritten to use only the following:
  - The empty string $\varepsilon$.
  - Individual characters
  - The operators $|$, $*$, and **string concatenation**.
Module 11.1: DFAs and regular expressions

Exercise: rewrite the following expressions to use only the options listed on the previous slide:

- a?
- a+
- a{3,5}
- \d
Module 11.1: DFAs and regular expressions

- **Proof**: by induction on the structure of the regular expression.

- Base cases:
  - The expression that matches the empty string:
    
  - The expression that matches no string:
Base cases (continued)

- The expression that matches a single character $a$:

\[ \begin{array}{c}
\text{Induction step:} \\
\circlearrowleft_a \\
\end{array} \]

- Consider a regular expression with $n$ characters.
- Suppose the theorem holds for every regular with fewer than $n$ characters.
- We consider three cases: the “last” operator could be $|$, string concatenation or $*$
• Induction step (continued):
  • The expression $E_1|E_2$ where $E_1$, $E_2$ are regular expressions:
Module 11.1: DFAs and regular expressions

- Induction step (continued and finished):
  - The expression $E_1E_2$ where $E_1$, $E_2$ are regular expressions:

  ![Diagram of $E_1E_2$]

  - The expression $E^*$ where $E$ is a regular expression:

  ![Diagram of $E^*$]
Module 11.1: DFAs and regular expressions

- Example: \((a|b)^*c\)
Module 11.1: DFAs and regular expressions

- How do we transform a NFA into a DFA?
  - A DFA that reads a string with $n$ characters ends up in exactly one state.
  - A NFA that reads a string with $n$ characters may end up in many different stages.
  - Can we figure out which states?
Module 11.1: DFAs and regular expressions

- Which state(s) will the following NFA end up in after reading the string \( ab \)?
  a) S1 only
  b) S6 only
  c) S3 or S7
  d) S4 or S6
  e) None of the above
Module 11.1: DFAs and regular expressions

- So we build the DFA as follows:
  - The DFA has one state for every *subset* of the states of the NFA (so $2^n$ states in total).
  - If the DFA is in state $\{ S_{i1}, S_{i2}, \ldots, S_{ik} \}$, and it sees a character $x$, then the new state is the state that contains every NFA state that we can get to from one of $S_{i1}, S_{i2}, \ldots, S_{ik}$ upon reading $x$.
  - A state of the DFA is accepting if it contains the accepting state of the NFA.
Module 11.1: DFAs and regular expressions

- Example: given the following NFA:

- We get the following partial DFA:
Module 11: Models of Computation

- Module Summary
  - (a bit of) Computing history.
  - A working computer.
  - DFAs and regular expressions.
  - **Computations that we are unable to perform.**
  - Appendix: working computer details.
Module 11.4: Computations we are unable to perform

- We have discussed several models of computation in the course:
  - Combinational circuits
  - Sequential circuits (the working computer).
  - DFAs

- One thing computer scientists (we) like to know about their (our) computational models is:
  - What can they do?
  - What can they not do?
Module 11.4: Computations we are unable to perform

- Example: DFAs
  - Intuition:
    - DFAs have no memory apart from the current state.
    - So a DFA with $n$ states can only “count” up to $n$ before it gets confused.
  - How do we formalize this?
    - We need to define a set $L$ of strings (language).
    - And show that no DFA can accept exactly the strings in $L$. 
Module 11.4: Computations we are unable to perform

- **Definition**: we denote by $a^n b^n$ a string that consists of
  - $n$ copies of the letter $a$, followed by
  - $n$ copies of the letter $b$.

The integer $n$ is not fixed and known ahead of time.

- **Theorem**: no DFA can recognize the language
  \[ \{ a^n b^n \mid n \in \mathbb{Z}^+ \} \].
Module 11.4: Computations we are unable to perform

- **Proof**: we use a proof by contradiction.
  - Suppose there is such a DFA. This DFA has $k$ states for some positive integer $k$.

Now look at what happens when the DFA is looking at the input $a^k b^k$.

While it's reading the input string, it goes through a number of states:
Module 11.4: Computations we are unable to perform

$q_0$ (initial state)  
$q_1$ (after reading the string $a$)  
$q_2$ (after reading the string $aa$)  
...  
$q_k$ (after reading the string $a^k$).

The DFA only has $k$ different states, so two of $q_0$, $q_1$, ..., $q_k$ must be the same state. Let us call these two $q_i$ and $q_j$, with $i < j$. 
Module 11.4: Computations we are unable to perform

**Observation 1**: If the DFA is in state $q_i$ and it sees $j - i$ copies of $a$ then it ends up in state

a) $q_0$

b) $q_i$

c) $q_k$

d) Some other state.
Module 11.4: Computations we are unable to perform

Observation 2: If the DFA is in state $q_i$ and it sees $k - i$ copies of $a$ then it ends up in state $\ldots$

a) $q_0$
b) $q_i$
c) $q_k$
d) Some other state.
Module 11.4: Computations we are unable to perform

What happens if we give the DFA the string $a^{k+(j-i)}b^k$?

While reading the first $i$ copies of $a$, it goes through states $q_0..q_i$.

Then it reads the next $j-i$ copies of $a$, and ends up in state $q_i$ again.

From $q_i$ it reads the next $k-i$ copies of $a$, and ends up in state $q_k$.

Then it reads the $k$ copies of $b$, and terminates in the same state it terminated when it was reading $a^kb^k$. 
Module 11.4: Computations we are unable to perform

But $a^k b^k$ should be accepted, and $a^{k+(j-i)} b^k$ should be rejected!

So the DFA will make a mistake on one of these two strings, which means it's not recognizing $\{ a^n b^n | n \in \mathbb{Z}^+ \}$ correctly.

This contradicts our initial assumption, and hence no DFA can recognize this language. QED
Module 11.4: Computations we are unable to perform

- Sequential circuits/Java/Racket are more powerful than DFAs.
  - For instance, you can easily write a Racket function or a Java method that will recognize \( \{ a^n b^n \mid n \in \mathbb{Z}^+ \} \).
- Can they solve every problem?
  - No: there are problems that can not be solved.

- **Halting Problem**: given a program \( P \) and an input \( I \), will \( P \) halt if we run it on input \( I \)?
Module 11.4: Computations we are unable to perform

- **Theorem**: It is not possible to write a program that solves the halting problem.

- **Proof 1**: proof by video:
  https://www.youtube.com/watch?v=92WHN-pAFCs
Module 11.4: Computations we are unable to perform

- Proof 2: we use a proof by contradiction.

Suppose this program exists.
Let us call it will-halt (Racket) or willHalt (Java).
We use this function or method to write the following function or method:
Module 11.4: Computations we are unable to perform

Racket version:

(define (paradox input)
  (if (will-halt input input)
      (paradox input) ; go into an infinite recursion
      true))

Java version:

public static void main(String[ ] args) {
  if (willHalt(args[0], args[0]))
    while(true) ;
  else
    return;
}
What happens when we call this program with \textit{itself} as input?

- If it halts, then \textit{will-halt/willHalt} returns true, and so it won't halt.
- But if it doesn't halt, then \textit{will-halt/willHalt} returns false, and so it will halt.

So whether the program halts or not, we end up with a contradiction.

Therefore this program does not exist. QED
CPSC 121: Models of Computation

😊 The End 😞
Module 11: Models of Computation

- Module Summary
  - (a bit of) Computing history.
  - A working computer.
  - DFAs and regular expressions.
  - Computations that we are unable to perform.
  - Appendix: working computer details.
Module 11.5: Appendix

- **Registers (32 bits each):**
  
<table>
<thead>
<tr>
<th>0</th>
<th>%eax</th>
<th>%esp</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>%ecx</td>
<td>%ebp</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>%edx</td>
<td>%esi</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>%ebx</td>
<td>%edi</td>
<td>7</td>
</tr>
</tbody>
</table>

- Instructions that only need one register use F for the second register.

- %esp is used as stack pointer.

- Memory contains $2^{32}$ bytes; all memory accesses load/store 32 bit words.
Module 11.5: Appendix

- Instruction types:
  - register/memory transfers:
    - rmmovl rA, D(rB) \[ M[D + R[rB]] \leftarrow R[rA] \]
      - Example: rmmovl %edx, 20(%esi)

  - mrmovl D(rB), rA \[ R[rA] \leftarrow M[D + R[rB]] \]
Module 11.5: Appendix

- Instruction types:
  - Other data transfer instructions
    - _rrmovl rA, rB_ \quad R[rB] \leftarrow R[rA]
    - _irmovl V, rB_ \quad R[rB] \leftarrow V
  - Arithmetic instructions
    - _addl rA, rB_ \quad R[rB] \leftarrow R[rB] + R[rA]
    - _subl rA, rB_ \quad R[rB] \leftarrow R[rB] - R[rA]
    - _andl rA, rB_ \quad R[rB] \leftarrow R[rB] \land R[rA]
    - _xorl rA, rB_ \quad R[rB] \leftarrow R[rB] \oplus R[rA]
Module 11.5: Appendix

- Instruction types:
  - Unconditional jumps
    - \texttt{jmp Dest} \quad \text{PC} \leftarrow \text{Dest}
  - Conditional jumps
    - \texttt{jle Dest} \quad \text{PC} \leftarrow \text{Dest if last result} \leq 0
    - \texttt{jl Dest} \quad \text{PC} \leftarrow \text{Dest if last result} < 0
    - \texttt{je Dest} \quad \text{PC} \leftarrow \text{Dest if last result} = 0
    - \texttt{jne Dest} \quad \text{PC} \leftarrow \text{Dest if last result} \neq 0
    - \texttt{jge Dest} \quad \text{PC} \leftarrow \text{Dest if last result} \geq 0
    - \texttt{jg Dest} \quad \text{PC} \leftarrow \text{Dest if last result} > 0
Module 11.5: Appendix

- Instruction types:
  - Conditional moves
  
  - cmovle rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} \leq 0 \]
  
  - cmovl  rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} < 0 \]
  
  - cmove  rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} = 0 \]
  
  - cmovne rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} \neq 0 \]
  
  - cmovge rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} \geq 0 \]
  
  - cmovg  rA, rB  \[ R[rB] \leftarrow R[rA] \text{ if last result} > 0 \]
Module 11.5: Appendix

- Instruction types:
  - Procedure calls and return support
    - call Dest $R[\%esp] \leftarrow R[\%esp]-4; M[R[\%esp]] \leftarrow PC; PC \leftarrow \text{Dest}$
    - ret $PC \leftarrow M[R[\%esp]]; R[\%esp] \leftarrow R[\%esp]+4$
    - pushl rA $R[\%esp] \leftarrow R[\%esp]-4; M[R[\%esp]] \leftarrow R[rA]$
    - popl rA $R[rA] \leftarrow M[R[\%esp]]; R[\%esp] \leftarrow R[\%esp]+4$
  - Others
    - halt
    - nop
# Module 11.5: Appendix

## Instructions format

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>0 0</td>
<td></td>
</tr>
<tr>
<td>nop</td>
<td>1 0</td>
<td></td>
</tr>
<tr>
<td>cmovXX rA, rB</td>
<td>2 fn rA rB</td>
<td></td>
</tr>
<tr>
<td>irmovl V, rB</td>
<td>3 0 F rB V</td>
<td></td>
</tr>
<tr>
<td>rmmovl rA, D(rB)</td>
<td>4 0 rA rB D</td>
<td></td>
</tr>
<tr>
<td>mrmovl D(rB), rA</td>
<td>5 0 rA rB D</td>
<td></td>
</tr>
<tr>
<td>OPI rA, rB</td>
<td>6 fn rA rB</td>
<td></td>
</tr>
<tr>
<td>jXX Dest</td>
<td>7 fn Dest</td>
<td></td>
</tr>
<tr>
<td>call Dest</td>
<td>8 0 Dest</td>
<td></td>
</tr>
<tr>
<td>ret</td>
<td>9 0</td>
<td></td>
</tr>
<tr>
<td>pushl rA</td>
<td>A 0 rA F</td>
<td></td>
</tr>
<tr>
<td>popl rA</td>
<td>B 0 rA F</td>
<td></td>
</tr>
</tbody>
</table>
Module 11.5: Appendix

- Instructions format:
  - Arithmetic instructions:
    - addl → fn = 0
    - andl → fn = 2
    - subl → fn = 1
    - xorl → fn = 3
  - Conditional jumps and moves:
    - jump → fn = 0
    - jl → fn = 2
    - jne → fn = 4
    - jle → fn = 1
    - je → fn = 3
    - jge → fn = 5
    - je → fn = 6