University of California, Berkeley
College of Engineering
Computer Science Division — EECS

Fall 2001 John Kubiatowicz

# Prerequisite Quiz

September 7, 2001
CS152 Computer Architecture and Engineering

This prerequisite quiz will be used in determining class admissions. The use of notes is not allowed during this quiz. Good Luck!

| | |
|---|---|
| Your Name: | Solution |
| SID Number: | |
| Discussion Section: | |

| | |
|---|---|
| 1 | 25 |
| 2 | 30 |
| 3 | 20 |
| 4 | 25 |
| Total | 100 |

[ This page left for $\pi$ ]

3.14159265358979323846264338327950288419716939937510582097494 4

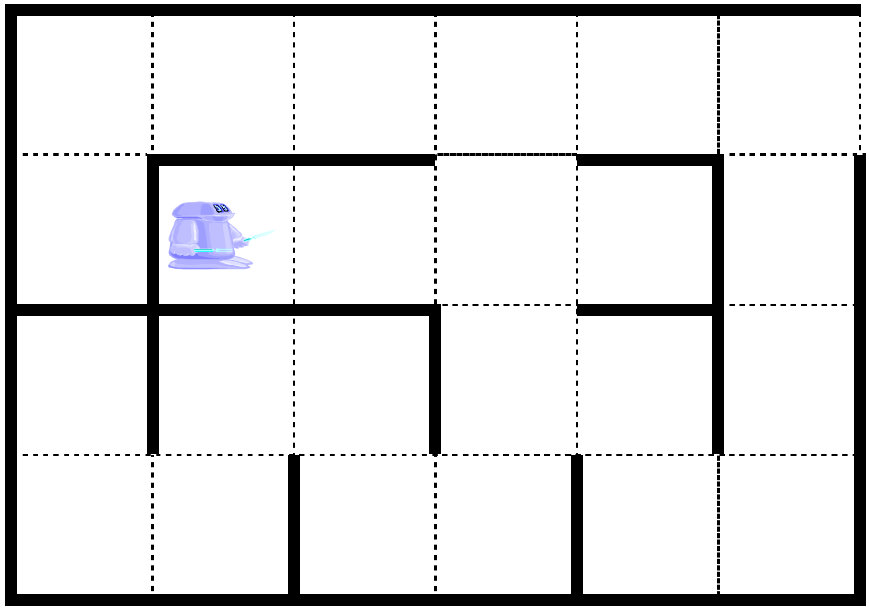
1) One way to calculate the square of a number **x** to add all the odd numbers from 1 to 2**x**-1. So $x^2 = (2x-1) + (2x-3) + ... + 3 + 1$. For example, $4^2 = 7 + 5 + 3 + 1 = 16$. Here is a recursive procedure to calculate the square of a number in C: ***Corrections below in blue.***

```
int square (int x) {
     if (x == 0) {
          return 0;
     } else if (x < 0) {
          return square (-x);
     } else {
          return ((2*x - 1) + square (x - 1));
     }
}
```

Here is a translation of the above program into MIPS assembly language (assume no branch delay slot):

```
square:  addi  $v0, $zero, 0        ; if a0 = 0, return 0
         beq   $a0, $zero, exit

         ; Don't really need recursive call for second clause!
         slt   $t0, $a0, $zero
         beq   $t0, $zero, notless  ; if a0 < zero jump ahead
         sub   $a0, $zero, $a0

notless: addi  $sp, $sp, -8
         sw    $s0, 4($sp)
         sw    $ra, 8($sp)          ; store $ra on stack, do NOT
                                    ; use offset 0 off of $sp
                                    ; by MIPS convention

         add   $t0, $a0, $a0
         addi  $s0, $t0, -1

         addi  $a0, $a0, -1
         jal   square

         add   $v0, $v0, $s0

         lw    $s0, 4($sp)

         lw    $s0, 4($sp)          ; restore s0, ra from stack
         lw    $ra, 8($sp)
         addi  $sp, $sp, -8

exit:    jr    $ra                  ; don't touch sp if a0 = 0
```

2) In this problem, you must design a finite state machine (FSM) to navigate a robot out of a maze. Assume that the robot is placed into the maze as shown in the Figure below:



Suppose that this robot can move forward one square, or turn right/left 90 degrees. Further, it has a single sensor that can detect the presence of a wall straight in front of it. Suppose we need to develop a state machine that is capable of controlling the robot. One possible technique is for the robot to "Follow the Left Wall", namely, keep its left side against a wall at all times. Following the right wall works as well.

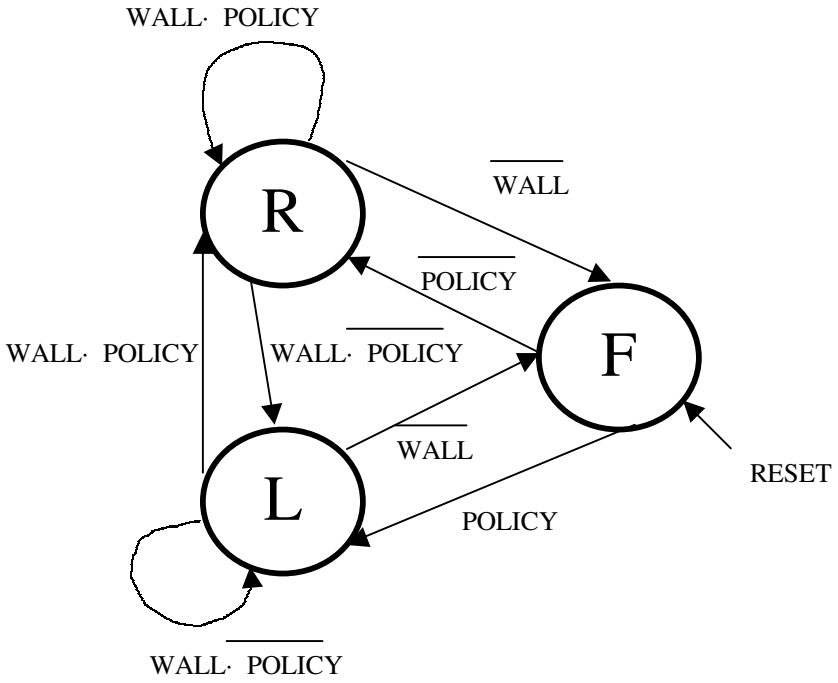
2a) In the above maze, how many *forward* steps will the robot take with the "Follow Left Wall" policy before it is standing outside? How about "Follow Right Wall"?

Counting the one step it takes to actually exit the maze, 14 for left wall policy and 24 for follow right wall policy.

2b) Explain why it *is not ok* for the robot to simply move forward as long as it is unblocked in the forward direction.

This is not a solution, because in some situations, like the one above, the robot will simply go back and forth and never find the exit.

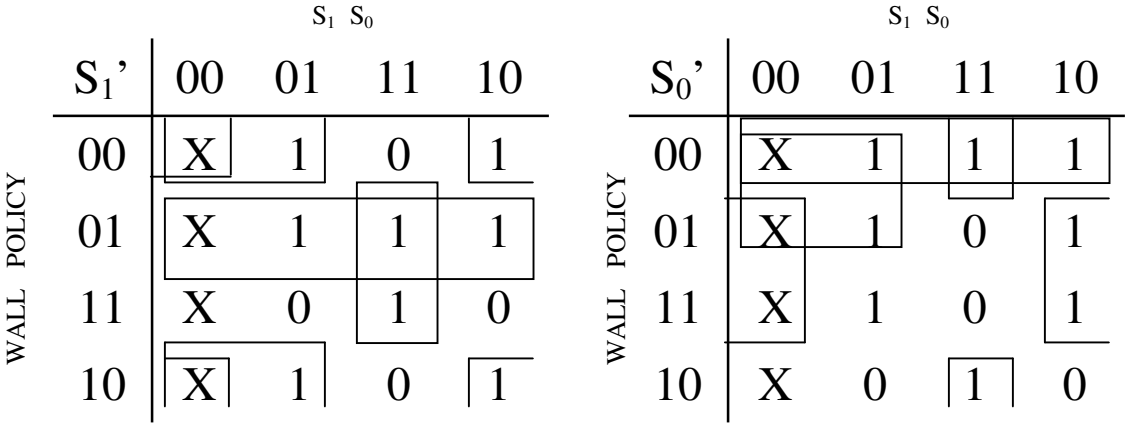
2c) Design a state machine to control the robot. There are 3 states, **Forward (F), Right (R),** and **Left (L)**, causing the robot to go forward or turn right or left respectively. Assume three inputs: "**WALL**" which is asserted if there is a wall in front of the robot; "**POLICY**" which is asserted for a "left wall" policy; and "**RESET**", which is asserted at the beginning. Assume that you can go forward immediately after **RESET.**.




2d) Construct a State Transition Table for this FSM. Assume the following two-bit encoding of states: **R=01**, **L=10**, and **F=11** Include the RESET signal in your table. Feel free to use "X" for inputs when appropriate.

| **WALL** | **POLICY** | **$S_1$** | **$S_0$** | **$S_1$'** | **$S_0$'** |
|---|---|---|---|---|---|
| X | 1 | 1 | 1 | 1 | 0 |
| X | 0 | 1 | 1 | 0 | 1 |
| 0 | X | 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | X | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 0 | 0 | 1 |

2e) Construct a Karnaugh map(s) to minimize the next-state logic equations. Derive the resulting simplified equations. You can ignore the RESET signal here.



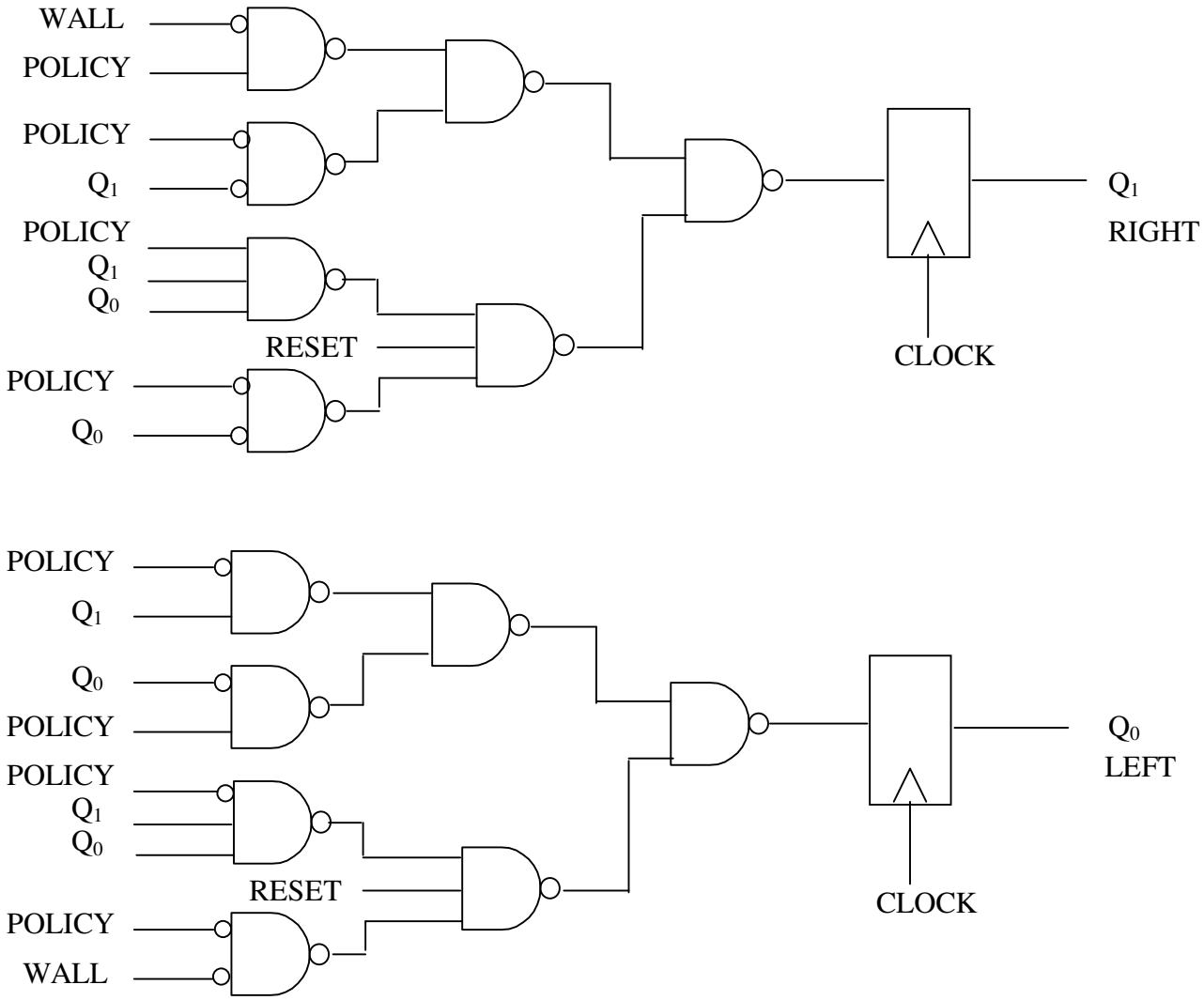

2f) Derive Next-State Logic Equations given the Karnaugh map. Include the RESET signal in your equations.

From the K-map:

$S_1' = \overline{WALL} \cdot POLICY + \overline{POLICY} \cdot \overline{S_1} + POLICY \cdot S_1 \cdot S_0 + \overline{POLICY} \cdot \overline{S_0} + RESET$

$S_0' = POLICY \cdot \overline{Q_1} + \overline{POLICY} \cdot Q_1 \cdot Q_0 + \overline{WALL} \cdot \overline{POLICY} + \overline{Q_0} \cdot POLICY + RESET$

2g) Implement the complete controller using only rising-edge triggered flip flops, inverters, and 2, 3, or 4-input NAND gates. Minimize the number of gates that you use. Assume that there are four inputs: **Clock, WALL, POLICY,** and **RESET.** Assume two controller outputs: **LEFT** to activate the left wheel (0=backward, 1=forward) and **RIGHT** to activate the right wheel (0=backward, 1=forward), i.e. turning left involves moving the LEFT wheel backwards and the right wheel forward.



[ This page left for scratch ]

Assume that we have a 16 bit system that uses signed, two's-complement integers. Perform the following conversions:

$-21_{10}$ to base 2: $(1111\ 1111\ 1110\ 1011)_2$

$E3A7_{16}$ to base 8: $161647_8$

$34_{10}$ to base 16: $22_{16}$

$(1111111100111100)_2$ to base 10: $-196_{10}$

$FFB3_{16}$ to base 10: $-77_{10}$

[ This page left for scratch ]

4) Assume that we have a 32-bit processor with 32-bit words, and that this processor is byte-addressed (i.e. addresses specify bytes). Now, suppose that this processor has a 32-word, *direct-mapped* cache with 2-word cache lines.

4a) Split the 32-bit address into "tag", "index", and "cache-line offset" pieces. We start you off with one of these.

tag: **bits 31 – 7**
index: **bits 6 – 3**
cache-line offset: **bits 2 – 0**

4b) Assume that the processor makes a series of byte accesses to memory (shown to the left below). Label each memory reference address as a Hit (H) or a Miss (M). Also, show the *final* contents of the cache in the table at the right. Fill in values for the TAG and words of the cache. Use the notation "M[i]" to mean the word in memory at address i (keep these word aligned!). Include no more than one value per word (words can be left blank).

| Reference Address | Hit or Miss? |
|---|---|
| $138_{10}(08A_{16})$: | M |
| $60_{10}(03C_{16})$: | M |
| $56_{10}(038_{16})$: | H |
| $140_{10}(08C_{16})$: | H |
| $419_{10}(1A3_{16})$: | M |
| $396_{10}(18C_{16})$: | M |
| $142_{10}(08E_{16})$: | M |
| $392_{10}(188_{16})$: | M |
| $175_{10}(0AF_{16})$: | M |
| $140_{10}(08C_{16})$: | M |
| $344_{10}(158_{16})$: | M |
| $487_{10}(1E7_{16})$: | M |
| $398_{10}(18E_{16})$: | M |
| $376_{10}(178_{16})$: | M |
| $380_{10}(17C_{16})$: | H |
| $160_{10}(0A0_{16})$: | M |

| Index | TAG Value | Word 0 | Word 1 |
|---|---|---|---|
| 0 | | | |
| 1 | 0x3 | M[0x188] | M[0x18C] |
| 2 | | | |
| 3 | | | |
| 4 | 0x1 | M[0x0A0] | M[0x0A4] |
| 5 | 0x1 | M[0x0A8] | M[0x0AC] |
| 6 | | | |
| 7 | 0x0 | M[0x038] | M[0x03C] |
| 8 | | | |
| 9 | | | |
| A | | | |
| B | 0x2 | M[0x158] | M[0x15C] |
| C | 0x3 | M[0x1E0] | M[0x1E4] |
| D | | | |
| E | | | |
| F | 0x2 | M[0x178] | M[0x17C] |

4c) Calculate the cache hit rate (you can leave as a fraction).

3/16

[ This page left for scratch ]

*[ The End ]*