CS61C Midterm 2 Review

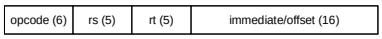
Summer 2004

Pooya Pakzad
Ben Huang
Navtej Sadhal

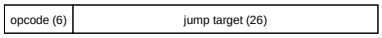
MIPS Instruction Formats

Two Examples: Branches and Jumps

1. Branches (I-format): How do you determine a branch address given a branch instruction?



2. j and jal (J-format): How do you determine a jump address given a branch instruction?



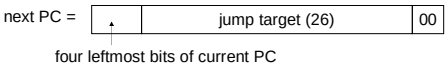
MIPS Instruction Formats

Two Examples: Branches and Jumps

1. Branches (I-format): How do you determine a branch address given a branch instruction?

   next PC = (PC + 4) + (<signed immediate> x 4)

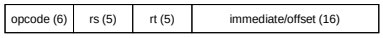
2. j and jal (J-format): How do you determine a jump address given a branch instruction?



MIPS Instruction Formats

One More Examples: Immediates

What is the difference between the immediate field of addiu and ori?



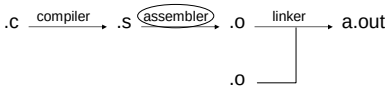
The immediate field of the addiu is signed whereas the immediate field of the ori is unsigned.

(How does this apply to the MIPS CPU you have been learning about?)

MIPS Pseudoinstructions

There are some assembly instructions in MIPS that don't have a 1-to-1 correlation with machine instructions.

- During the assembly stage, pseudoinstructions are translated into non-pseudoinstructions.



Non-Pseudoinstructions are known as TAL instructions.

MIPS Pseudoinstructions

Translate all pseudoinstructions to TAL instructions.

```
            .data
x:          .word 2

            .text
main:       la      $t0, x
            lw      $t0, 0($t0)
loop:       addiu   $t1, $t0, 0x1FFFF
            mul     $t0, $t0, $t1
            li      $t2, 0x1FFFF
            beq     $t0, $t2, loop
            li      $t2, 1
```

MIPS Pseudoinstructions

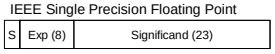
Translate all pseudoinstructions to TAL instructions.

```
            .data
x:          .word 2

            .text
main:       lui     $t0, l.x        # la    $t0, x
            ori     $t0, $t0, r.x
            lw      $t0, 0($t0)
loop:       lui     $at, 0x1        # addiu $t1, $t0, 0x1FFFF
            ori     $at, 0xFFFF
            addu    $t1 $t0, $at
            mult    $t0, $t1        # mul   $t0, $t0, $t1
            mflo    $t0
            lui     $t2, 0x1        # li    $t2, 0x1FFFF
            ori     $t2, 0xFFFF
            beq     $t0, $t2, loop
            addiu   $t2, $0, 1      # li    $t2, 1
```

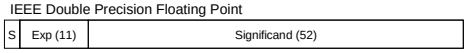
IEEE Floating Point Representation

IEEE Single Precision Floating Point



$(-1)^S$ x (1.Significand) x $2^{Exp-127}$

IEEE Double Precision Floating Point



$(-1)^S$ x (1.Significand) x $2^{Exp-1023}$

Why is there a bias? Where is there an assumed 1?

Floating Point Thought Questions

For IEEE Single/Double Precision Floating Point Rep:

- How many numbers can you represent?
- What is the smallest/largest positive numbers you can represent?
- How do you compare two floating point numbers?
- How do you add/subtract two floating point numbers?

Floating Point Thought Questions

For IEEE Single/Double Precision Floating Point Rep:

- True or False: For every 32-bit integer, there is a floating point number that exactly equals that integer (and vice versa).
- What is the largest integer you can cast into a floating point number and back again, and still get the same value? (integer → floating point → integer)
- Etc…

Floating Point Thought Questoins

For IEEE Single/Double Precision Floating Point Rep:

- Also consider:
  - Changing IEEE floating point representation?
  - Inventing a new 8-bit floating point representation?
  - Changing the number of bits in the significand and exponent?

Floating Point Values

Fill in the table for the value of each row (given the exponent and significand):

Exponent	Significand	Value
11111111	0	+infinity
11111111	Not 0	NaN
Anything Else	Anything	normalized
00000000	0	+zero
00000000	Not 0	denormalized

Normalized: $(-1)^S$ x (1.Significand) x $2^{Exp-Bias}$
Denormalized: $(-1)^S$ x (0.Significand) x 2^{1-Bias}

Boolean Algebra

- Combinational Logic
- Truth Tables
- Sum of Products
- Algebraic Simplification
- Programmable Logic Arrays

Truth Tables

- Construct a truth table for a 3 input, 1 output logic function that determines if the majority of the bits are 0.

Input	Output
000	1
001	1
010	1
011	0
100	1
101	0
110	0
111	0

Sum of Products

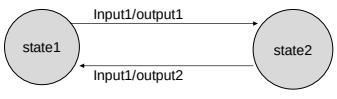
- To find the sum of products, you AND together the bits of each line that has 1 on the output and then OR the terms together.
- Find the sum of products for the previous function:
  S = A'B'C' + A'B'C + A'BC' + AB'C'

Simplify using Boolean Algebra

- S = A'B'C' + A'B'C + A'BC' + AB'C'
- S = A'B' + A'B' + B'C'

Finite State Machines

- FSMs contain a finite number of states, inputs and outputs.
- Can be represented on with a state transition diagram:

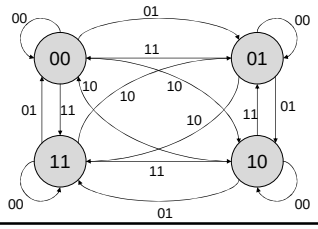


Finite State Machines

- Outputs:
  - State determined: output(currentState); outputs can be marked on states (Moore Machine)
  - State and input determined: output(currentState, input); outputs marked on transition arcs (Mealey Machine)

Finite State Machine

- Construct a state transition diagram for a 2 bit accumulator that takes a 2 bit input. It will wrap back around on overflow.



Finite State Machines

- Is the output state-determined? **Yes**
- Write a truth table for the nextState function

curState	Input	nextState	curState	Input	nextState
00	00	00	10	00	10
00	01	01	10	01	11
00	10	10	10	10	00
00	11	11	10	11	01
01	00	01	11	00	11
01	01	10	11	01	00
01	10	11	11	10	01
01	11	00	11	11	10

Verilog

- Hardware *description* language
  - Time is important; everything happens in parallel
  - Pure structural Verilog has no concept of sequence or procedure. It is only a description of hardware

Structural Verilog

- Write a 3 bit parity check module using only primitive AND and OR gates in structural Verilog; include 1ns gate delay:
  - List inputs and outputs (module header):

```
module parity(S, A, B, C);
  input A, B, C;
  output S;
```

  - Find the boolean equation:
    - P = A'B'C + A'BC' + AB'C' + ABC

Structural Verilog

```
module parity(S, A, B, C);
  input A, B, C;
  output S;
  wire w0, w1, w2, w3;

  and #1
    (w0, ~A, ~B, C), (w1, ~A, B, ~C),
    (w2, A, ~B, ~C), (w3, A, B, C);
  or #1
    (S, w0, w1, w2, w3);
endmodule
```

Is there a simpler way?

```
  xor(S, A, B, C);
```

Adding State

- Write a module that takes 3 bits of data and 1 parity bit on every clock cycle and checks if the parity matches (error checking). If ever two mismatches in a row occur, the failure bit (output) goes high immediately. Include 1ns gate delay on each gate. You may use only structural Verilog and assume a DFF module exists:

```
module DFF(CLK, RST, Q, D);
//clk to Q time is 1ns
```

Adding State

```
module parityChecker(CLK, RST, data, parity, fail);
  input [2:0] data;
  input parity;
  output fail;
  wire w0, w1, w2;

  xor #1 (w0, data[2], data[1], data[0]);
  xor #1 (w1, w0, parity);
  DFF myff (.CLK(CLK), .RST(RST), .Q(w2), .D(w1));
  and #1 (fail, w2, w1);
endmodule
```

Verilog testbench

- Write a testbench for the parityChecker:

```
module testParityChecker;
      reg [2:0] data;
      reg parity, fail_exp, CLK=0, RST=1;
      wire fail;

      parityChecker(CLK, RST, data, parity, fail);

      initial repeat(12) #10 CLK=~CLK;

      initial begin
               RST=1;
               #15 RST=0; data=3'b101; parity=0; fail_exp=0;
               #20 data=3'b111; parity=1; fail_exp=0;
               #20 data=3'b000; parity=1; fail_exp=0;
               #20 data=3'b110; parity=1; fail_exp=1;
               #20 data=3'b011; parity=0; fail_exp=0;
      end
      initial repeat(6) begin
               #14 $display("data: %b, parity:%b, fail; %b, fail_exp:
                     %b", data, parity, fail, fail_exp);
               #6
               end
endmodule2
```

Dataflow Verilog

- Dataflow Verilog uses continuous assigns. It has the same effect as structural verilog.
  - Left side of assignment must be a wire
  - Right side can be any signal
  - Behavioral operators are allowed on right side
  - Syntax is like an assignment in C
    - But behavior is like structural Verilog!

Dataflow Verilog

- Write the original parity module using dataflow Verilog without using ^:

```
module parity(S, A, B, C);
   input A, B, C;
   output S;

   assign S = (~A && ~B && C) || (~A && B && ~C) ||
          (A && ~B && ~C) || (A && B && C);

endmodule
```

Dataflow Verilog

- What is a simple way to create a multiplexor on the fly using dataflow Verilog?

```
assign X = S ? B : A
```