“What’s with all these 1s and 0s?”

David Jacobs

1001 0010 0000 1000

1111 1111 1111 1111
They’re a two’s complement integer!

“What’s with all these 1s and 0s?”

1001 0010 0000 1000

1111 1111 1111 1111

Invert bits and add 1

0110 1101 1111 0111

0000 0000 0000 0001

(-1) \times (6 \times 16^7 + 11 \times 16^6 + 15 \times 16^5 + 7 \times 16^4 + 0 \times 16^3 + 0 \times 16^2 + 0 \times 16^1 + 1 \times 16^0)

= -181349505
They’re a floating point number!

“What’s with all these 1s and 0s?”

<table>
<thead>
<tr>
<th>Exponent</th>
<th>Significand</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>nonzero</td>
<td>Denorm</td>
</tr>
<tr>
<td>1-254</td>
<td>anything</td>
<td>+/- fl. pt. #</td>
</tr>
<tr>
<td>255</td>
<td>0</td>
<td>+/- ∞</td>
</tr>
<tr>
<td>255</td>
<td>nonzero</td>
<td>NaN</td>
</tr>
</tbody>
</table>

Expressed in binary

\[ (-1)^1 \times 1.00010001111 \ldots \times 2^{(36-127)} = -4.323 \times 10^{(-28)} \]
"What’s with all these 1s and 0s?"

They’re a MIPS instruction!

It’s an I-type!

According to your green sheet...

opcode 36 → lbu $rt, imm($rs)

$16 is $s0 and $8 is $t0

lbu $s0, -1($t0)
They’re 32 separate logical values!

“What’s with all these 1s and 0s?”

The disk isn’t ready to be read.

I showered today

The stove is on

Interrupts are enabled

Fall 2006 CS61C Final Review, David Poll, David Jacobs, Michael Le
If there’s one thing you learn...

\[ N \text{ bits can represent } 2^N \text{ things} \]
C and Memory

Get an \( n \)-element array of things

\[
\text{array} = (\text{thing} *) \\
\text{malloc} (n*\text{sizeof(thing)})
\]

Don’t forget to free it later.

\[
\text{free(array)};
\]
typedef struct node {
    int value;
    struct node* next;
} ent;

stack push(stack s, int val) {
    // reserve space
    // set values
    // return new node
}

typedef ent * stack;

int peek(stack s) {
    return value
}

stack pop(stack s, int * val) {
    // change
    // free entry on top
    // return the next one
}
typedef struct node {
    int value;
    struct node* next;
} ent;

stack push(stack s, int val) {
    ent * new = (ent *) malloc(sizeof(ent));
    new->value = val;
    new->next = s;
    return new;
}

int peek(stack s) {
    return s->value;
}

stack pop(stack s, int * val) {
    ent * temp = s->next;
    *val = s->value;
    free(s);
    return temp;
}
Memory Management

- **First fit**
  - Allocate the first available chunk big enough
- **Next fit**
  - Allocate the first chunk after the last one allocated
- **Best fit**
  - Allocate the smallest chunk capable of satisfying the request
Memory Management

- **Free List**
  - Linked list of free chunks, use first/next/best fit

- **Slab Allocator**
  - Fixed number of $2^n$ sized chunks, can use a bitmap to track. Free list for larger requests.

- **Buddy Allocator**
  - $2^n$ chunks can merge with their “buddy” to make a $2^{(n+1)}$ chunk. Free list for larger requests.
Automatic Memory Management

- Reference Counting
  - Keep track of pointers to each malloc’d chunk. Free when references = 0.

- Mark and Sweep
  - Recursively follow “root set” of pointers, marking accessible chunks. Free unreachable chunks in place.

- Copying
  - Split memory into two pieces. Mark reachable chunks as above, then copy and defragment into other half.
MIPS

Sum: `addiu $sp, $sp, -8`
`sw $ra, 0($sp)`
`sw $s0, 4($sp)`
`add $s0, $a0, $0`
`addiu $a0, $a0, -1`
`jal Sum`
`add $v0, $v0, $s0`
`lw $s0, 4($sp)`
`lw $ra, 0($sp)`
`addiu $sp, $sp, 8`
`jr $ra`

Prologue

Body

Epilogue

Saved registers

Argument registers

Return value

Return address
typedef struct node {
    int value; // offset 0
    struct node* next; // offset 4
} ent;

stack push(stack s, int val){
    ent * new = (ent *) malloc(sizeof(ent));
    new->value = val;
    new->next = s;
    return new;
}
typedef struct node {
    int value; // offset 0
    struct node* next; //offset 4
} ent;

stack push(stack s, int val){
    ent * new = (ent *) malloc(sizeof(ent));
    new->value = val;
    new->next = s;
    return new;
}
CALL

C program: foo.c

Compiler

Assembly program: foo.s

Assembler

Object (mach lang module): foo.o

Linker

Executable (mach lang pgm): a.out

Loader

Memory

Fall 2006 CS61C Final Review, David Poll, David Jacobs, Michael Le
Ok, I get it. But how does it work?!
Problem

You have been hired to build a plate spinning controller for a robot. The robot can only handle the following orientations of a plate:

lean left, balanced, lean right, broken

In addition, there is a wind factor:

strong left, left, right, strong right

Depending on the situation, the robot will respond by pushing the plate left or right, spin the plate, or do nothing.

How would you begin designing this circuit?
Finite State Machine
A general approach to designing one

1. Identify the states
   - orientation

2. Identify the inputs
   - wind, current orientation, action

3. Identify the outputs
   - run through every scenario

4. Identify the transitions
Finite State Machine
A general approach to designing one

1. Identify the states
2. Identify the inputs
3. Identify the outputs
4. Identify the transitions

Orientation
Wind, Current State
Action, Next State
Find all possible combos
Additional Problem Information

The robot will respond as follows:

• If the wind is strong, the orientation shifts two steps.
  • This means, \textbf{Left} + \textbf{Strong Left} = \textbf{Broken Plate}

• Robot spins plate only when plate returns to the \textbf{balanced} state due to the wind

• Once plate is \textbf{broken}, controller does nothing
What does the FSM look like?

Inputs
- CurrentState
  - Left, Right, Balance, Broken
- Wind
  - Strong Left, Left, Right, Strong Right

Outputs
- NextState
  - Left, Right, Balance, Broken
- Action
  - Push left, spin, push right, do nothing

The robot will respond as follows:
- If the wind is strong, the orientation shifts two steps.
  - This means, **Left + Strong Left** is a **Broken Plate**
- Robot spins plate only when plate returns to the **balanced** state due to the wind
- Once plate is **broken**, controller does nothing
Solution to the FSM
What Next?

- Now that we have an FSM, what do we do now?
Building Truth Tables

Two general methods

- Running through every combination of the inputs
- If an input/output is multiple bits, break treat each bit as an individual input
- Follow all the transition arcs of your FSM
Solution to the Truth Table

<table>
<thead>
<tr>
<th>State</th>
<th>Wind</th>
<th>Act</th>
</tr>
</thead>
<tbody>
<tr>
<td>10 - L</td>
<td>10 - S, L</td>
<td>01 - push right</td>
</tr>
<tr>
<td>11 - C</td>
<td>11 - S, R</td>
<td>10 - push left</td>
</tr>
<tr>
<td>01 - R</td>
<td>00 - L</td>
<td>01 - R</td>
</tr>
<tr>
<td>00 - B</td>
<td>01 - L</td>
<td>00 - nothing</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Curr1</th>
<th>Curr0</th>
<th>Wind1</th>
<th>Wind0</th>
<th>Next1</th>
<th>Next0</th>
<th>Out1</th>
<th>Out0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<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>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Going from Truth Table to Circuit

- Canonical Sums of Products
  - For each output, OR every combination that produces a true value
  - Each combination depends on AND’ed inputs
  - Commonly known as the Brute-Force method

- For example, for majority circuit

\[
\text{Maj}(A, B, C) = A \cdot \overline{B} + B \cdot C + \overline{A} \cdot C + A \cdot B \cdot C
\]
Develop your Expressions

- Using your truth table, determine the expressions for Next₁, Next₀, Act₁, and Act₀.
Brute Force Result

Next_1 = \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0}

Next_0 = \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0}

Act_1 = \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0}

Act_0 = \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0} + \overline{C_1C_0} \overline{W_1W_0}
Reflecting on Brute Force

- Easy, but ugly.
- Sometimes not the optimal solution
- What can we do to get a more elegant result?
Boolean Algebra: Elegant Solution

Use Boolean Algebra and simplify your expressions!

<table>
<thead>
<tr>
<th>x \cdot \overline{x} = 0</th>
<th>x + \overline{x} = 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>x \cdot 0 = 0</td>
<td>x + 1 = 1</td>
</tr>
<tr>
<td>x \cdot 1 = x</td>
<td>x + 0 = x</td>
</tr>
<tr>
<td>x \cdot x = x</td>
<td>x + x = x</td>
</tr>
<tr>
<td>x \cdot y = y \cdot x</td>
<td>x + y = y + x</td>
</tr>
<tr>
<td>(xy)z = x(yz)</td>
<td>(x + y) + z = x + (y + z)</td>
</tr>
<tr>
<td>x(y + z) = xy + xz</td>
<td>x + yz = (x + y)(x + z)</td>
</tr>
<tr>
<td>xy + x = x</td>
<td>(x + y)x = x</td>
</tr>
<tr>
<td>\overline{x \cdot y} = \overline{x} + \overline{y}</td>
<td>(x + y) = \overline{x} \cdot \overline{y}</td>
</tr>
</tbody>
</table>

complementarity  
laws of 0’s and 1’s identities  
idempotent law  
commutativity  
associativity  
distribution  
uniting theorem  
DeMorgan’s Law
Elegant Solution

Next_1 = \overline{C_1}C_0 \overline{W_1}W_0 + \overline{C_1}C_0 \overline{W_1}W_0 + \overline{C_1}C_0 W_1 \overline{W_0} + \
C_1 \overline{C_0} \overline{W_1}W_0 + C_1 \overline{C_0} W_1 \overline{W_0} + C_1 C_0 \overline{W_1}W_0

Next_0 = \overline{C_1}C_0 W_1 \overline{W_0} + \overline{C_1}C_0 W_1 \overline{W_0} + \overline{C_1}C_0 \overline{W_1}W_0 + \
C_1 \overline{C_0} \overline{W_1}W_0 + C_1 \overline{C_0} W_1 \overline{W_0} + C_1 C_0 \overline{W_1}W_0

Brute Force

Simplified

Next_1 = \overline{C_1}C_0 W_1 + \overline{C_1}C_0 W_0 + C_1 \overline{C_0} W_1 + C_1 W_1 \overline{W_0}

Next_0 = C_1 C_0 W_1 + C_0 \overline{W_1}W_0 + C_1 \overline{W_1}W_0 + C_1 C_0 W_0
Elegant Solution

\[
\begin{align*}
\text{Act}_1 &= \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + \overline{C}_1 C_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 \overline{W}_1 W_0 \\
\text{Act}_0 &= \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 \overline{W}_1 W_0 + C_1 \overline{C}_0 W_1 W_0
\end{align*}
\]

Brute Force

Simplified

\[
\begin{align*}
\text{Act}_1 &= \overline{C}_1 C_0 \overline{W}_1 + C_1 \overline{C}_0 W_1 W_0 \\
\text{Act}_0 &= \overline{C}_1 C_0 \overline{W}_1 \overline{W}_0 + C_1 \overline{C}_0 W_1
\end{align*}
\]
Expressions to Gates

- With your expressions, conversion to gates is mechanical using the sums of products approach
  - Each term becomes an AND gate
  - Collect the output of the appropriate AND gate into an OR
Next circuit
The remaining circuits...

- They are quite trivial and I’m sure you didn’t want me to draw them for you.

- Don’t forget about registers! They are used to hold state.
Master SDS is all about mastering the Trifecta®

It is possible to transition from any state to any other state. However, the ease of this transition is dependent on the complexity of the problem.
Single Cycle CPU Design
Tasks a CPU must do

- Fetch an instruction
- Decode the instruction
  - Get values from registers and set control lines
- Execute instruction (arithmetic)
- Meddle with Memory, if necessary
- Record result of instruction
  - a.k.a. register write back
Building/Extending a CPU Datapath

1. Determine what function you want to do
   - I want to support adding of two registers

2. Determine what you have to work with
   - I have registers, muxes, gates, and lots of wires

3. Formulate a plan of bringing data from where it is found to where it is needed
   - I need to move data from registers to an ALU

4. Execute your plan

5. Determine Control Signals
Applying Those Steps

If I have the following C code:

```c
*p = z + 4;
```

Converting it to MIPS would produce

```mips
addi $t0, z, 4
sw $t0, 0(p)
```

Let’s suppose you want to do this in 1 instruction
Step 1 – Determine function
Step 1 – Determine function

I want to add two values and store them into memory
Step 1 – Determine function

I want to add two values and store them into memory

As a guidance, let’s layout what the datapath must do

\[ \text{Mem}[R[rs]] = R[rt] \oplus \text{SignExtImmed} \]
Step 2 – Determine what is available
Step 2 – Determine what is available
Step 3 – Formulate Plan
Step 3 – Formulate Plan

1. Add R[rt] to SignExtImmed
2. Send R[rt]+SignExtImmed to Memory Data
3. Send R[rs] to Memory Addr
Step 4 – Execute Plan

<table>
<thead>
<tr>
<th>MemDataSrc</th>
<th>Data In</th>
</tr>
</thead>
<tbody>
<tr>
<td>MemAddrSrc</td>
<td>MemtoReg</td>
</tr>
</tbody>
</table>

Fall 2006 CS61C Final Review, David Poll, David Jacobs, Michael Le
## Step 5 – Set Control Lines

<table>
<thead>
<tr>
<th>Control</th>
<th>Value</th>
<th>Control</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>nPC_sel</td>
<td>normal</td>
<td>ExtOp</td>
<td>Sign</td>
</tr>
<tr>
<td>RegDst</td>
<td>X</td>
<td>MemWr</td>
<td>1</td>
</tr>
<tr>
<td>RegWrite</td>
<td>0</td>
<td>MemToReg</td>
<td>X</td>
</tr>
<tr>
<td>ALUCtrl</td>
<td>X</td>
<td>MemDataSrc</td>
<td>1</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>X</td>
<td>MemAddrSrc</td>
<td>1</td>
</tr>
</tbody>
</table>
Things to Keep in Mind

- There is more than one way to modify datapath to produce same result

- If you split a line leading into an input, you need to use a mux.
  - Send original line into 0
  - Send new line into 1
Pipelining
Pipelining Problems

• Hazards
  • Structural: Using some type of circuit two different ways, at the same time
  • Data: Instruction depends on result of prior instruction
  • Control: Later instruction fetches delayed to wait for result of branch
Solving Hazards

- **Structural**
  - add hardware, use other properties

- **Control**
  - do things earlier such as with branches
    - delay slot compromise

- **Data**
  - use forwarding, interlocking at worst case

\[\text{CPU forced to stop despite forwarding}\]
Data Dependencies and Forwarding

- **Data Dependency**
  - Needing data at decode when updated data has not reached register write back

- **Forwarding**
  - moving data from one stage to another
  - Exception is R to D – not considered forwarding because no new wire is laid down
Two methods for determining data dependencies and forwarding

1. If arrows are drawn starting from end (right side) of R to stage where data is needed in a later instruction, then the arrow represents **data dependency**

2. If arrows are drawn starting from when data is first available (right side of stage) to where data is absolutely needed (left side of stage), arrow represents **data dependency and forwarding possibility**
Arrow Drawing Guidelines (for method 2)

- Only draw arrow only if R of updated value of register does not line up on top to the left of D
- Arrows should never span more than 3 instructions (red arrow bad)

```
addi $t0, $t0, 0     F  D  A  M  R
add  $t1, $t1, $t0      F  D  A  M  R
sub  $t2, $t1, $a0         F  D  A  M  R
and  $t3, $t0, $a1            F  D  A  M  R
ori  $t4, $t0, $t1               F  D  A  M  R
```
Pitfalls in arrow drawing

- Pay attention to how registers are used
  - Not all instructions update registers (i.e. sw)

- Some instructions use registers two different ways
  - lw/sw uses one register for address, the other for data

- Method #1 generally has arrows going left
  - Arrow going to the right means no data dependency

- Method #2 generally has arrows going right;
  - Arrow going to the left for #2 means forwarding won’t help; meaning you must stall the pipeline (i.e. do interlock)
Branch Delay Slot

- Any instruction that follows a branch instruction occupies that slot

- That instruction is executed 100% of the time, unless we have advanced pipelining logic (pipeline flushing, out of order execution, etc).

- Unless we tell you otherwise, there is NO advanced pipeline logic.
Infamous Example

How many clock cycles would it take to run the following code at left, if the pipelined MIPS CPU had all solutions to control and data hazards as discussed in class (branch delay slot, load interlock, register forwarding)?

```
addi $1, $0, 2

loop: add $0, $0, $0
       beq $1, $0, done
       add $4, $3, $2
       add $5, $4, $3
       add $6, $5, $4
       addi $1, $1, -1
       beq $0, $0, loop
       addi $1, $1, -1

done: beq $0, $0, exit
       addi $1, $0, 3

exit: addi $1, $0, 1
```
Infamous Example

    addi $1, $0, 2  1
    loop: add  $0, $0, $0  2, 10
            beq  $1, $0, done  3, 11
            add  $4, $3, $2  4, 12
            add  $5, $4, $3  5
            add  $6, $5, $4  6
            addi $1, $1, -1  7
            beq  $0, $0, loop  8
            addi $1, $1, -1  9
    done: beq  $0, $0, exit  13
            addi $1, $0, 3  14
    exit: addi $1, $0, 1  15, 16, 17, 18, 19
Infamous Example

```
addi $1, $0, 2
loop: add $0, $0, $0
beq $1, $0, done
add $4, $3, $2
done: beq $0, $0, exit
add $5, $4, $3
add $6, $5, $4
addi $1, $1, -1
beq $0, $0, loop
addi $1, $1, -1
exit: addi $1, $0, 1
```

19 Cycles

Pipeline Drain
More Pipelining Practice

- How many cycles are needed to execute the following code:

```assembly
loop:
[1] add $a0, $a0, $t1
[2] lw $a1, 0($a0)
[3] add $a1, $a1, $t1
[4] sw $a1, 0($t1)
[5] add $t1, $t1, -1
[6] bne $0, $0, end
[7] add $t9, $t9, 1
```

- CPU has
  - no forwarding units
  - will interlock on any hazard
  - no delayed branch
  - 2nd stage branch compare
  - instructions are not fetched until compare happens
  - memory CAN be read/written on the same cycle
  - same registers CAN be read/written on the same cycle
More Pipelining Practice

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>F</td>
<td>D</td>
<td>A</td>
<td>M</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[3]</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[4]</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[5]</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[6]</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[7]</td>
<td>F</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

18 Cycles
More Pipelining Practice

- How many cycles are needed to execute the following code:

```
loop:
[1]  add $a0, $a0, $t1
[2]  lw  $a1, 0($a0)
[3]  add $a1, $a1, $t1
[4]  sw  $a1, 0($t1)
[5]  add $t1, $t1, -1
[6]  bne $0, $0, end
[7]  add $t9, $t9, 1
```

- CPU has
  - all forwarding units
  - will interlock on any hazard
  - delayed branch
  - 2nd stage branch compare

memory CAN be read/written on the same cycle

- same registers CAN be read/written on the same cycle
## More Pipelining Practice

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1]</td>
<td>F</td>
<td>D</td>
<td>A</td>
<td>M</td>
<td>R</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

12 Cycles
What else?

David Poll
Caches

- Why?
- TIO
- Write-back
- Write-through
- Replacement
- Hit/Miss

Image from HowStuffWorks.com
Example

VM
- 1 MiB Virtual Memory Space, 32 KiB Physical Memory
- 4 KiB Page Size
  - 0x0000C
  - 0x200D0
  - 0x10000
  - 0x202D0
  - 0x200D8
  - 0x204D0

Cache
- 32 KiB Addressable Memory, 1 KiB Cache Size, 128 B Block Size, LRU Replacement, 2-way set associative
  - 0x000C
  - 0x10D0
  - 0x2000
  - 0x12D0
  - 0x10D8
  - 0x14D0

Fall 2006 CS61C Final Review, David Poll, David Jacobs, Michael Le
VM

- Why?
- VPN vs. PPN
- Page Fault
- Page in, Page out

Image from HowStuffWorks.com
VM/Caches

- What happens when we switch processes?
- Problem with Page Tables? (where are they?)
- AMAT
  - AMAT = Hit Time + (Miss %) x (AMAT for Miss)
  - Give an expression for AMAT of a system with VM (with TLB) and Cache
Performance

- CPU Time (CPI)
- Example:
  - Memory Read – 10%, CPI = 18
  - Memory Write – 15%, CPI = 20
  - ALU – 30%, CPI = 1
  - Branch – 45%, CPI = 2
  - Overall CPI?
  - CPU Speed = 1 GHz, 1 Million instructions, CPU Time?
  - Cache added. Memory Read/Write halved. Improvement?

Megahertz Myth
- What determines performance?
I/O

- Polling
  - Are we there yet?
- Interrupts
  - Wake me when we get there.
- Memory Mapped I/O
Networks

- Sharing vs. Switching
- Half-duplex vs. Full-duplex
- Packets
  - Header
  - Payload
  - Trailer
- Ack?
- TCP/IP

**Packet - E-mail Example**

<table>
<thead>
<tr>
<th>Packet Component</th>
<th>Details</th>
<th>Length</th>
</tr>
</thead>
<tbody>
<tr>
<td>Header</td>
<td>Sender's IP address, Receiver's IP address, Protocol, Packet number</td>
<td>96 bits</td>
</tr>
<tr>
<td>Payload</td>
<td>Data</td>
<td>896 bits</td>
</tr>
<tr>
<td>Trailer</td>
<td>Data to show end of packet, Error correction</td>
<td>32 bits</td>
</tr>
</tbody>
</table>
Disks

Latency:
- Seek Time + Rotation Time + Transfer Time + Controller Overhead
RAID

- RAID-0
  - Striped
- RAID-1
  - Mirrored
- RAID-4
  - Striped, parity drive
- RAID-5
  - Striped, striped parity
Parallelization

- Why?
- Distributed Computing
- Parallel Processing
- Amdahl’s law
  - Time $\geq s + 1/p$
  - Speedup $\leq 1/s$
Conclusion

Questions on the Sp-04 Final?