Intel has developed a new technique to generate random numbers that is suitable for integration directly into the CPU! This circuit can turn out 2.4 billion random numbers a second!

---

**Review**

- **In MIPS Assembly Language:**
  - Registers replace variables
  - One Instruction (simple operation) per line
  - Simpler is Better, Smaller is Faster
- **New Instructions:**
  - `add`, `addi`, `sub`
- **New Registers:**
  - C Variables: `$s0 - $s7`
  - Temporary Variables: `$t0 - $t7`
  - Zero: `$zero`

---

**Assembly Operands: Memory**

- C variables map onto registers; what about large data structures like arrays?
- 1 of 5 components of a computer: memory contains such data structures
- But MIPS arithmetic instructions only operate on registers, never directly on memory.
- Data transfer instructions transfer data between registers and memory:
  - Memory to register
  - Register to memory

---

**Anatomy: 5 components of any Computer**

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, and then transfer back to memory when done.

---

**Data Transfer: Memory to Reg (1/4)**

- To transfer a word of data, we need to specify two things:
  - Register: specify this by `# ($0 - $31)` or symbolic name (`$s0, ..., $t0, ...`)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address.
    - Other times, we want to be able to offset from this pointer.
- Remember: “Load FROM memory”

---

**Data Transfer: Memory to Reg (2/4)**

- To specify a memory address to copy from, specify two things:
  - A register containing a pointer to memory
  - A numerical offset (in bytes)
- The desired memory address is the sum of these two values.
- Example: 8 ($t0)
  - specifies the memory address pointed to by the value in $t0, plus 8 bytes
Data Transfer: Memory to Reg (3/4)

- **Load Instruction Syntax:**
  
  1. 2,3(4)
  
  - where
    1) operation name
    2) register that will receive value
    3) numerical offset in bytes
    4) register containing pointer to memory

- **MIPS Instruction Name:**
  
  - **lw** (meaning Load Word, so 32 bits or one word are loaded at a time)

Data Transfer: Memory to Reg (4/4)

- **Example:**
  
  `lw $t0,12($s0)`

  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then load the value from the memory pointed to by this calculated sum into register `$t0`

- **Notes:**
  
  - `$s0` is called the base register
  - 12 is called the offset
  - offset is generally used in accessing elements of array or structure: base reg points to beginning of array or structure (note offset must be a constant known at assembly time)

Data Transfer: Reg to Memory

- **Also want to store from register into memory**
  
  - Store instruction syntax is identical to Load's

- **MIPS Instruction Name:**
  
  - **sw** (meaning Store Word, so 32 bits or one word is stored at a time)

- **Example:**
  
  `sw $t0,12($s0)`

  This instruction will take the pointer in `$s0`, add 12 bytes to it, and then store the value from register `$t0` into that memory address

- **Remember:** “Store INTO memory”

Pointers v. Values

- **Key Concept:** A register can hold any 32-bit value. That value can be a (signed) int, an unsigned int, a pointer (memory addr), and so on
  
  - E.g., if you write: add $t2,$t1,$t0
    - then $t0$ and $t1$ better contain values that can be added
  - E.g., if you write: lw $t2,0($t0)
    - then $t0$ better contain a pointer

- **Don’t mix these up!**

Addressing: Byte vs. Word

- **Every word in memory has an address,** similar to an index in an array
- **Early computers numbered words like C numbers elements of an array:**
  
  - Memory[0], Memory[1], Memory[2], ...
- **Computers needed to access 8-bit bytes as well as words (4 bytes/word)**
- **Today machines address memory as bytes, i.e., “Byte Addressed” hence 32-bit (4 byte) word addresses differ by 4**
  
  - Memory[0], Memory[4], Memory[8]

C vs MIPS: Memory Accesses

- **What offset in lw to select A[5] in C?**
  
  - A is an int array
  - 4x5=20 to select A[5]: byte v. word
- **Compile by hand using registers:**
  
  - `g = h + A[5];`
    - `g: $s1, h: $s2, $s3: base address of A`
  - **1st transfer from memory to register:**
    - `lw$t0,20($s3) # $t0 gets A[5]`
  - Add 20 to $s3 to select A[5], put into $t0
  - Next add it to h and place in g
    - `add $s1,$s2,$t0 # $s1 = h+A[5]`
Notes about Memory

- Pitfall: Forgetting that sequential word addresses in machines with byte addressing do not differ by 1.
  - Many an assembly language programmer has toiled over errors made by assuming that the address of the next word can be found by incrementing the address in a register by 1 instead of by the word size in bytes.
  - Also, remember that for both `lw` and `sw`, the sum of the base address and the offset must be a multiple of 4 (to be word aligned).

More Notes about Memory: Alignment

- MIPS requires that all words start at byte addresses that are multiples of 4 bytes

<table>
<thead>
<tr>
<th>Last hex digit of address is:</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aligned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Not Aligned</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Called Alignment: objects fall on address that is multiple of their size

Administrivia

- Project 1 due on Saturday
- Next few lectures are pretty packed
  - Be Prepared!
- No Section on Monday. See the section notes online for what would have been discussed
- Should I hold OH on Monday?
- Midterm:
  - Friday, July 16th, 9:30am-12:30pm (Room TBA)
- Final:
  - Thursday, Aug 12th, 8am-11:00am (Room TBA)

Role of Registers vs. Memory

- What if more variables than registers?
  - Compiler tries to keep most frequently used variable in registers
  - Less common variables in memory: spilling
- Why not keep all variables in memory?
  - Smaller is faster: registers are faster than memory
  - Registers more versatile:
    - MIPS arithmetic instructions can read 2, operate on them, and write 1 per instruction
    - MIPS data transfer only read or write 1 operand per instruction, and no operation

So Far...

- All instructions so far only manipulate data...we've built a calculator of sorts.
- In order to build a computer, we need ability to make decisions...
- C (and MIPS) provide labels to support "goto" jumps to places in code.
  - C: Horrible style; MIPS: Necessary!
- Heads up: pull out some papers and pens, you'll do an in-class exercise!

C Decisions: if Statements

- 2 kinds of `if` statements in C
  - `if (condition) clause`
  - `if (condition) clause1 else clause2`
- Rearrange 2nd `if` into following:
  - `if (condition) goto L1; clause2; goto L2; L1: clause1; L2:`
  - Not as elegant as `if-else`, but same meaning. NEVER WRITE C CODE LIKE THIS
MIPS Decision Instructions

- Decision instruction in MIPS:
  
  ```
  beq register1, register2, L1
  ```
  
  is “Branch if (registers are) equal”
  
  Same meaning as (using C):
  
  ```
  if (register1==register2) goto L1
  ```

- Complementary MIPS decision instruction
  
  ```
  bne register1, register2, L1
  ```
  
  is “Branch if (registers are) not equal”
  
  Same meaning as (using C):
  
  ```
  if (register1!=register2) goto L1
  ```

- Called conditional branches

MIPS Goto Instruction

- In addition to conditional branches, MIPS has an unconditional branch:
  
  ```
  j label
  ```
  
  Called a Jump Instruction: jump (or branch) directly to the given label without needing to satisfy any condition

  ```
  if  (register1==register2) goto L1
  ```

- Technically, it's the same effect as:
  
  ```
  beq $0,$0,label
  ```
  
  since it always satisfies the condition.

Compiling C if into MIPS (1/2)

- Compile by hand
  
  ```
  if (i == j) f=g+h;
  else f=g-h;
  ```

- Use this mapping:
  
  ```
  f: $s0
  g: $s1
  h: $s2
  i: $s3
  j: $s4
  ```

- Exit

Compiling C if into MIPS (2/2)

- Compile by hand
  
  ```
  if (i == j) f=g+h;
  else f=g-h;
  ```

- Final compiled MIPS code:
  
  ```
  beq $s3,$s4, True # branch i==j
  sub $s0,$s1,$s2 # f=g-h (false)
  j Fin # goto Fin
  True: add $s0,$s1,$s2 # f=g+h (true)
  Fin:
  ```

  Note: Compiler automatically creates labels to handle decisions (branches). Generally not found in HLL code.

Loading, Storing bytes 1/2

- In addition to word data transfers (lw, sw), MIPS has byte data transfers:
  
  ```
  load byte: lb
  store byte: sb
  ```

- same format as lw, sw

- E.g., lb $s0, 3($s1)
  
  contents of memory location with address = sum of “3” + contents of register s1 is copied to the low byte position of register s0.

Loading, Storing bytes 2/2

- What do with other 24 bits in the 32 bit register?
  
  ```
  xxxx xxxx xxxx xxxx xxxx xxxx
  ```

  - is copied to “sign-extend”

  byte loaded

  This bit

- Normally don’t want to sign extend chars

- MIPS instruction that doesn’t sign extend when loading bytes:
  
  ```
  load byte unsigned: lbu
  ```
Overflow in Arithmetic (1/2)

- Reminder: Overflow occurs when there is a mistake in arithmetic due to the limited precision in computers.
- Example (4-bit unsigned numbers):

  15
  + 3
  18

  But we don’t have room for 5-bit solution, so the solution would be 0010, which is +2, and wrong.

Overflow in Arithmetic (2/2)

- Some languages detect overflow (Ada), some don’t (C)
- MIPS solution is 2 kinds of arithmetic instructs:
  - These cause overflow to be detected
    - add (add)
    - add immediate (addi)
    - subtract (sub)
  - These do not cause overflow detection
    - add unsigned (addu)
    - add immediate unsigned (addiu)
    - subtract unsigned (subu)

Compiler selects appropriate arithmetic

Two “Logic” Instructions

- Here are 2 more new instructions
- Shift Left: sll $s1,$s2,2
  - $s1= 4*I
- Shift Right: srl

Loops in C/Assembly (1/3)

- Simple loop in C; A[] is an array of ints
  
  do { g = g + A[i];
       i = i + j;
  } while (i != h);

- Rewrite this as:

  Loop: g = g + A[i];
  i = i + j;
  if (i != h) goto Loop;

- Use this mapping:

  g, h, i, j, base of A
  $s1, $s2, $s3, $s4, $s5

Loops in C/Assembly (2/3)

- Final compiled MIPS code:

  Loop: sll $t1,$s3,2 # $t1= 4*I
  addu $t1,$t1,$s5 # $t1=addr A+i
  lw $t1,0($t1) # $t1=A[i]
  addu $s1,$s1,$t1 # g=g+A[i]
  addu $s3,$s3,$s4 # i=i+j
  bne $s3,$s2,Loop # goto Loop
  # if i!=h

- Original code:

  Loop: g = g + A[i];
  i = i + j;
  if (i != h) goto Loop;

Loops in C/Assembly (3/3)

- There are three types of loops in C:
  - while
  - do... while
  - for

  - Each can be rewritten as either of the other two, so the method used in the previous example can be applied to these loops as well.

  - Key Concept: Though there are multiple ways of writing a loop in MIPS, the key to decision-making is conditional branch
Peer Instruction

We want to translate $x = y$ into MIPS
($x, y$ ptrs stored in: $s0$ $s1$)

1: add $s0$, $s1$, zero
2: add $s1$, $s0$, zero
3: lw $s0$, 0($s1) a) 1 or 2
4: lw $s1$, 0($s0) b) 3 or 4
5: lw $t0$, 0($s1) c) 5→6
6: sw $t0$, 0($s0) d) 6→5
7: lw $s0$, 0($t0) e) 7→8
8: sw $s1$, 0($t0)

“And in Conclusion…”

- Memory is byte-addressable, but lw and sw access
  one word at a time.
- A pointer (used by lw and sw) is just a memory
  address, we can add to it or subtract from it (using
  offset).
- A Decision allows us to decide what to execute at
  run-time rather than compile-time.
- C Decisions are made using conditional statements
  within if, while, do while, for.
- MIPS Decision making instructions are the
  conditional branches: beq and bne.
- One can store and load (signed and unsigned)
  bytes as well as words with lb, lbu.
- Unsigned add/sub don’t cause overflow
- Loops using beq and bne.
- New Instructions:
  lw, sw, beq, bne, j, lb, sb, lbu, addu, addiu, subu, srl, sll … WOW