IBM has developed a new supercomputer that is cooled by 140° (F) water. It uses 40% less power than traditional systems, and its waste heat is used to warm the building.

**Review**
- Simplifying MIPS: Define instructions to be same size as data word (one word) so that they can use the same memory (compiler can use `lw` and `sw`).
- Computer actually stores programs as a series of these 32-bit numbers.
- MIPS Machine Language Instruction: 32 bits representing a single instruction

### R-Format Example (1/2)
- Pseudocode (OPERATION on green sheet)
  \[ \text{add} \quad R[rd] = R[rs] + R[rt] \]
- MIPS Instruction:
  \[ \text{add} \quad $8,9,10 \]
  
  \[ \begin{array}{c}
  \text{opcode} = 0 \quad \text{(look up on green sheet)} \\
  \text{funct} = 32 \quad \text{(look up on green sheet)} \\
  \text{rd} = 8 \quad \text{(destination)} \\
  \text{rs} = 9 \quad \text{(first operand)} \\
  \text{rt} = 10 \quad \text{(second operand)} \\
  \text{shamt} = 0 \quad \text{(not a shift)}
  \end{array} \]

### R-Format Example (2/2)
- MIPS Instruction:
  \[ \text{add} \quad $8,9,10 \]
  
  \[ \begin{array}{cccccccc}
  \text{Decimal number per field representation:} & 0 & 9 & 10 & 8 & 0 & 32 \\
  \text{Binary number per field representation:} & 000000 & 01001 & 01010 & 01000 & 100000 & 01024 \ \\
  \text{hex representation:} & 012A & 4020_{\text{Hex}} \\
  \text{decimal representation:} & 19,546,144_{\text{bin}}
  \end{array} \]

Called a Machine Language Instruction
(This is part of the process of assembly)

### I-Format Instructions (1/4)
- What about instructions with immediates?
  - 5-bit field only represents numbers up to the value 31: immediates may be much larger than this
  - Ideally, MIPS would have only one instruction format (for simplicity): unfortunately, we need to compromise
- Define new instruction format that is partially consistent with R-format:
  - First notice that, if instruction has immediate, then it uses at most 2 registers.

### I-Format Instructions (2/4)
- Define “fields” of the following number of bits
  \[ 6 + 5 + 5 + 16 = 32 \text{ bits} \]
  
  \[ \begin{array}{cccc}
  \text{opcode} & \text{rs} & \text{rt} & \text{immediate} \\
  \end{array} \]
  
  - Again, each field has a name:
    - Key Concept: Three fields are consistent with R-Format instructions. Most importantly, `opcode` is still in same location.
I-Format Instructions (3/4)

- What do these fields mean?
  - **opcode**: same as before except that, since there's no **funct** field, *opcode* uniquely specifies an instruction in I-format.
  - This also answers question of why R-format has two 6-bit fields to identify instruction instead of a single 12-bit field: in order to be consistent as possible with other formats while leaving as much space as possible for immediate field.
  - **rs**: specifies a register operand (if there is one)
  - **rt**: specifies register which will receive result of computation (this is why it's called the *target register* "rt") or other operand for some instructions.

I-Format Example (1/2)

- Pseudocode (OPERATION on green sheet)
  ```
  addi \text{R}[rt] = \text{R}[rs] + \text{SignExtImm}
  ```

- MIPS Instruction:
  ```
  addi \$21, \$22, -50
  ```
  - *opcode* = 8 (look up on green sheet)
  - *rs* = 22 (register containing operand)
  - *rt* = 21 (target register)
  - *immediate* = -50 (by default, this is decimal)

I-Format Example (2/2)

- MIPS Instruction:
  ```
  addi \$21, \$22, -50
  ```
  - **Decimal/field representation:**
    - 8
    - 22
    - 21
    - -50
  - **Binary/field representation:**
    - 001000 10110 10101 1111111111001110
  - **hexadecimal representation:** 22D5
  - **decimal representation:** 584, 449, 998

Peer Instruction

Which instruction has same representation as 35<sub>ten</sub>?

- a) add $0, $0, $0
- b) subu $s0,$s0,$s0
- c) lw $0, 0($0)
- d) addi $0, $0, 35
- e) subu $0, $0, $0

Registers numbers and names:
- 0: $0, .. 8: $8, 9:$t0, .. 15: $t7, 16: $s0, 17: $s1, .. 23: $s7

Opcodes and function fields (if necessary)
- **add**: opcode = 0, funct = 32
- **subu**: opcode = 0, funct = 35
- **addi**: opcode = 8
- **lw**: opcode = 35

I-Format Instructions (4/4)

- The Immediate Field:
  - *addi, slti, sltiu*, the immediate is sign-extended to 32 bits. Thus, it's treated as a signed integer.
  - 16 bits \(\rightarrow\) can be used to represent immediate up to \(2^{16}\) different values.
  - This is large enough to handle the offset in a typical *lw* or *sw*, plus a vast majority of values that will be used in the *slti* instruction.
  - We'll see what to do when the number is too big later today....

MIPS Instruction:

```
addi $21, $22, -50
```
I-Format Problems (0/3)

- Problem 0: Unsigned # sign-extended?
  - addiu, sltiu, sign-extends immediates to 32 bits. Thus, # is a “signed” integer.

Rationale
- addiu so that can add w/ out overflow. Remember, the u means don’t signal overflow, not signed vs unsigned integers!
- sltiu suffers so that we can have easy HW
  - Does this mean we’ll get wrong answers?
  - Nope, it means assembler has to handle any unsigned immediate \( 2^{16} \leq n < 2^{16} \) (i.e., with a 1 in the 15th bit and 0s in the upper 2 bytes) as it does for numbers that are too large. ⇒

I-Format Problem (1/3)

- Problem:
  - Chances are that addi, lw, sw and slti will use immediates small enough to fit in the immediate field.
  - …but what if it’s too big?
  - We need a way to deal with a 32-bit immediate in any I-format instruction.

I-Format Problem (2/3)

- Solution to Problem:
  - Handle it in software + new instruction
  - Don’t change the current instructions: instead, add a new instruction to help out

  New instruction:
  - lui register, immediate
  - stands for Load Upper Immediate
  - takes 16-bit immediate and puts these bits in the upper half (high order half) of the register
  - sets lower half to 0s

I-Format Problems (3/3)

- Solution to Problem (continued):
  - So how does lui help us?
  - Example:
    - addi $t0, $t0, 0xABABCDCD
    - …becomes
    - lui $at, 0xABAB
    - ori $at, $at, 0xCDCD
    - add $t0, $t0, $at
  - Now each I-format instruction has only a 16-bit immediate.
  - Wouldn’t it be nice if the assembler would this for us automatically? (later)

Branches: PC-Relative Addressing (1/5)

- Use I-Format

<table>
<thead>
<tr>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>opcode specifies beq versus bne</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rs and rt specify registers to compare</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>What can immediate specify?</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>immediate is only 16 bits</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PC (Program Counter) has byte address of current instruction being executed; 32-bit pointer to memory</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>So immediate cannot specify entire address to branch to.</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Branches: PC-Relative Addressing (2/5)

- How do we typically use branches?
  - Answer: if-else, while, for
  - Loops are generally small: usually up to 50 instructions
  - Function calls and unconditional jumps are done using jump instructions (j and jal), not the branches.

Conclusion: may want to branch to anywhere in memory, but a branch often changes PC by a small amount
Branches: PC-Relative Addressing (3/5)

- Solution to branches in a 32-bit instruction: PC-Relative Addressing
- Let the 16-bit immediate field be a signed two's complement integer to be added to the PC if we take the branch.
- Now we can branch ±215 bytes from the PC, which should be enough to cover almost any loop.
- Any ideas to further optimize this?

Branches: PC-Relative Addressing (4/5)

- Note: Instructions are words, so they're word aligned (byte address is always a multiple of 4, which means it ends with 00 in binary).
  - So the number of bytes to add to the PC will always be a multiple of 4.
  - So specify the immediate in words.
- Now, we can branch ±215 words from the PC (or ±217 bytes), so we can handle loops 4 times as large.

Branches: PC-Relative Addressing (5/5)

- Branch Calculation:
  - If we don’t take the branch:
    \[ \text{PC} = \text{PC} + 4 = \text{byte address of next instruction} \]
  - If we do take the branch:
    \[ \text{PC} = (\text{PC} + 4) + (\text{immediate} \times 4) \]
- Observations
  - Immediate field specifies the number of words to jump, which is simply the number of instructions to jump.
  - Immediate field can be positive or negative.
  - Due to hardware, add immediate to (PC+4), not to PC; will be clearer why later in course

Branch Example (1/3)

- MIPS Code:
  
  Loop:
  beq $9,$0, End
  addu $8,$8,$10
  addiu $9,$9,-1
  j Loop
  End:

- Immediate field:
  - Number of instructions to add to (or subtract from) the PC, starting at the instruction following the branch.
  - In beq case, immediate = 3

Branch Example (2/3)

- MIPS Code:
  
  Loop:
  beq $9,$0, End
  addu $8,$8,$10
  addiu $9,$9,-1
  j Loop
  End:

- immediate Field:
  
  decimal representation:
  4 9 0 3
  binary representation:
  000100 01001 00000 000000000000011

Branch Example (3/3)

- MIPS Code:
  
  Loop:
  beq $9,$0, End
  addu $8,$8,$10
  addiu $9,$9,-1
  j Loop
  End:

- decimal representation:
  4 9 0 3
Questions on PC-addressing

- Does the value in branch immediate field change if we move the code?
- What do we do if destination is > $2^{15}$ instructions away from branch?
- Why do we need different addressing modes (different ways of forming a memory address)? Why not just one?

J-Format Instructions (1/5)

- For branches, we assumed that we won’t want to branch too far, so we can specify change in PC.
- For general jumps (j and jal), we may jump to anywhere in memory.
- Ideally, we could specify a 32-bit memory address to jump to.
- Unfortunately, we can’t fit both a 6-bit opcode and a 32-bit address into a single 32-bit word, so we compromise.

J-Format Instructions (2/5)

- Define two “fields” of these bit widths:
  
  \[
  \begin{array}{c|c}
  31 & \text{6 bits} \\
  26 & \text{26 bits} \\
  0 &
  \end{array}
  \]

- As usual, each field has a name:

  - \text{opcode}
  - \text{target address}

- Key Concepts
  - Keep \text{opcode} field identical to R-format and I-format for consistency.
  - Collapse all other fields to make room for large target address.

J-Format Instructions (3/5)

- For now, we can specify 26 bits of the 32-bit bit address.
- Optimization:
  - Note that, just like with branches, jumps will only jump to word aligned addresses, so last two bits are always 00 (in binary).
  - So let’s just take this for granted and not even specify them.

J-Format Instructions (4/5)

- Now specify 28 bits of a 32-bit address
- Where do we get the other 4 bits?
  - By definition, take the 4 highest order bits from the PC.
  - Technically, this means that we cannot jump to anywhere in memory, but it’s adequate 99.9999…% of the time, since programs aren’t that long
    - only if straddle a 256 MB boundary
  - If we absolutely need to specify a 32-bit address, we can always put it in a register and use the \text{jr} instruction.

J-Format Instructions (5/5)

- Summary:
  - New PC = (PC+4)[31..28], target address, 00
  - Understand where each part came from!
- Note: \{ , , \} means concatenation
  - \{ 4 bits , 26 bits , 2 bits \} = 32 bit address
    - \{ 1010, 11111111111111111111111111, 00 \} = 10101111111111111111111111111100
  - Note: Book uses ||
When combining two C files into one executable, recall we can compile them independently & then merge them together. When merging two or more binaries:

1) Jump insts don't require any changes.
2) Branch insts don't require any changes.

### In conclusion
- MIPS Machine Language Instruction:
  - 32 bits representing a single instruction

<table>
<thead>
<tr>
<th>R</th>
<th>opcode</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>opcode</td>
<td>rs</td>
<td>rt</td>
<td></td>
<td></td>
<td>immediate</td>
</tr>
<tr>
<td>J</td>
<td>opcode</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>target address</td>
</tr>
</tbody>
</table>

- Branches use PC-relative addressing.
- Jumps use absolute addressing.
- Disassembly is simple and starts by decoding opcode field. (more in a week)

### Bonus slides
- These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.
- The slides will appear in the order they would have in the normal presentation

#### lui ... ori example
- Here is the contents of $t0 after a lui, then an ori

```
lui $t0, 0xABAB
1010 1011 1010 1011 0000 0000 0000 0000
ori $t0, $t0, 0xCDCD
1010 1011 1010 1011 1100 1101 1100 1101
```

### Peer Instruction Question

When combining two C files into one executable, recall we can compile them independently & then merge them together. When merging two or more binaries:

1) Jump insts don't require any changes.
2) Branch insts don't require any changes.

<table>
<thead>
<tr>
<th>12</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>a)</td>
<td>FF</td>
</tr>
<tr>
<td>b)</td>
<td>FT</td>
</tr>
<tr>
<td>c)</td>
<td>TF</td>
</tr>
<tr>
<td>d)</td>
<td>TT</td>
</tr>
<tr>
<td>e)</td>
<td>dunno</td>
</tr>
</tbody>
</table>