## Charging for Instructions

<table>
<thead>
<tr>
<th>Instr</th>
<th>200</th>
<th>50</th>
<th>100</th>
<th>200</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>I-fetch</td>
<td>Reg Access</td>
<td>ALU</td>
<td>Reg Access</td>
<td>400</td>
</tr>
<tr>
<td>LW</td>
<td>I-fetch</td>
<td>Reg Access</td>
<td>ALU</td>
<td>Mem Access</td>
<td>Reg Access</td>
</tr>
<tr>
<td>SW</td>
<td>I-fetch</td>
<td>Reg Access</td>
<td>ALU</td>
<td>Mem Access</td>
<td>550</td>
</tr>
<tr>
<td>BEQ</td>
<td>I-fetch</td>
<td>Reg Access</td>
<td>ALU</td>
<td></td>
<td>350</td>
</tr>
<tr>
<td>J</td>
<td>I-fetch</td>
<td></td>
<td></td>
<td></td>
<td>200</td>
</tr>
</tbody>
</table>
Instruction Mix

• Cycle Time = 600 ps
• Execute billion instructions = .6 s
• For GCC instruction mix
  0.22 * 600 +
  0.11 * 550 +
  0.49 * 400 +
  0.16 * 350 +
  0.02 * 200 = 448.5
• Billion GCC instructions = .45 s
Drawbacks of single cycle implementation

• All instructions take the same time although
  – some instructions are longer than others;
    • e.g. load is longer than add since it has to access data
      memory in addition to all the other steps that add does
  – thus the “cycle” has to be for the “longest path”

• Some combinational units must be replicated since
  used in the same cycle
  – e.g., ALU for computing branch address and ALU for
    computing branch outcome
    • but this is no big deal (these duplicate resources will be
      needed when we will pipeline instructions)
Alternative to single cycle

• Have a shorter cycle and instructions execute in multiple (shorter) cycles
• The (shorter) cycle time determined by the longest delay in individual functional units (e.g., memory or ALU etc.)
• Possibility to streamline some resources since they will be used at different cycles
• Since there is need to keep information “between cycles”, we’ll need to add some stable storage (registers) not visible at the ISA level
• Not all instructions will require the same number of
Multiple cycle implementation

• Follows the decomposition of the steps for the execution of instructions
  – **Cycle 1.** Instruction fetch and increment PC
  – **Cycle 2.** Instruction decode and read source registers and branch address computation
  – **Cycle 3.** ALU execution or memory address calculation or set PC if branch successful
  – **Cycle 4.** Memory access (load/store) or write register (arith/log)
  – **Cycle 5** Write register (load)
• Note that branch takes 3 cycles, load takes 5 cycles, all others take 4 cycles
Instruction fetch

• Because fields in the instruction are needed at different cycles, the instruction has to be kept in stable storage, namely an Instruction Register (IR)
• The register transfer level actions during this step
  \[ IR \leftarrow \text{Memory}[PC] \]
  \[ PC \leftarrow PC + 4 \]
• Resources required
  – Memory (but no need to distinguish between instruction and data memories; later on we will because the need will reappear when we pipeline instructions)
  – Adder to increment PC
  – IR
Instruction decode and read source registers

- Instruction decode: send opcode to control unit and...(see later)
- Perform “optimistic” computations that are not harmful
  - Read rs and rt and store them in non-ISA visible registers A and B that will be used as input to ALU
    
    \[
    A \leftarrow \text{REG}[\text{IR}[25:21]] \text{ (read rs)} \\
    B \leftarrow \text{REG}[\text{IR}[20:16]] \text{ (read rt)}
    \]
  
  - Compute the branch address just in case we had a branch!
    \[
    \text{ALUout} \leftarrow \text{PC} + (\text{sign-ext}(\text{IR}[15:0]) \times 4 \text{ (ALUout is also a non-ISA visible register)}
    \]
- New resources
  - A, B, ALUout
ALU execution

- If instruction is R-type
  \[ \text{ALUout} \leftarrow A \text{ op. } B \]
- If instruction is Immediate
  \[ \text{ALUout} \leftarrow A \text{ op. sign-extend(IR[15:0])} \]
- If instruction is Load/Store
  \[ \text{ALUout} \leftarrow A + \text{sign-extend(IR[15:0])} \]
- If instruction is branch
  If (A=B) then \( PC \leftarrow \text{ALUout} \) (note this is the ALUout computed in the previous cycle)
- No new resources
Memory access or ALU completion

- If Load
  \[ MDR \leftarrow \text{Memory[ALUout]} \] (MDR is the Memory Data Register non-ISA visible register)

- If Store
  \[ \text{Memory[ALUout]} \leftarrow B \]

- If arith
  \[ \text{Reg[IR[15:11]]} \leftarrow \text{ALUout} \]

- New resources
  - MDR
Load completion

- Write result register
  \[\text{Reg}[\text{IR}[20:16]] \leftarrow \text{MDR}\]
Streamlining of resources

• No distinction between instruction and data memory
• Only one ALU
• But a few more muxes and registers (IR, MDR etc.)
FloorPlan for Multicycle MIPS