Exercise
Write the code segment

\[ \begin{align*}
\text{result} &= \text{operand}_1 + \text{operand}_2 \\
\text{x} &= \text{y} \\
\text{z} &= \text{w}
\end{align*} \]

- In stack architecture
- In accumulator architecture
- In register-memory architecture
- In load-store architecture
- In memory-memory architecture

GPR Architecture Classification: By Number Of Operands

- Three
  
  \[
  \begin{array}{ccc}
  \text{result} & \text{operand}_1 & \text{operand}_2
  \end{array}
  \]

- Two
  
  \[
  \begin{array}{cc}
  \text{result} & \text{operand}_1 & \text{operand}_2
  \end{array}
  \]

GPR Architecture Classification: By Number Of Memory References

- Three operands
  
  \[
  \text{register}, \text{memory}, \text{memory-memory}, \text{load-store}, \text{one operand in memory}, \text{register-memory}
  \]

- Two operands
  
  \[
  \text{memory-memory}, \text{one operand in memory}, \text{register-memory}
  \]

Which Architecture Is the Best?

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register-memory</td>
<td>Simple instruction length instruction</td>
<td>Higher instruction count</td>
</tr>
<tr>
<td>Register-memory</td>
<td>Fixed length instruction</td>
<td>Longer programs</td>
</tr>
<tr>
<td>Register-memory</td>
<td>Similar CPI</td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Compiler optimizations</td>
<td></td>
</tr>
<tr>
<td>Register-memory</td>
<td>Better instruction density</td>
<td>Source operand is destroyed</td>
</tr>
<tr>
<td>Register-memory</td>
<td>Best instruction density</td>
<td>Longer instructions</td>
</tr>
<tr>
<td>Register-memory</td>
<td>Most significant byte first</td>
<td>CPI vary by operand location</td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Least significant byte first</td>
<td>CPI vary by operand location</td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Best instruction density</td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Largest instructions</td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Most significant byte first</td>
<td></td>
</tr>
<tr>
<td>Memory-memory</td>
<td>Least significant byte first</td>
<td></td>
</tr>
</tbody>
</table>

Memory Addressing

- When we read from/write to a memory, how big piece of data can we handle in one try?
- Least addressable unit is byte (8 bits)
- Also accessible: half-word (16 bits), word (32 bits), double word (64 bits)
- How are longer objects stored?

```
“time” —— 74 69 6D 65
Big endian  74 69 6D 65
0 1 2 3
Most significant byte first

Little endian 65 6D 69 74
0 1 2 3
Least significant byte first
```

Exercise

- How would we store the string “time is now!” if the addressing is:
  
  - Big endian
  - Little endian

```
74 69 6D 65 20 68 73 20 6E 6F 77 21
```
Memory Addressing

- Accesses to objects larger than 1 byte are aligned
  - If object size is \( s \)
  - Object's address \( A \) will be chosen so that \( A \mod s = 0 \)

Addressing Modes

- **Register**: operand is in register
  - Add R4, R3
  - Second operand's value = Regs[R3]

- **Immediate**: operand is constant
  - Add R4, #3
  - Second operand's value = #3

Addressing Modes

- **Register indirect**: operand is in memory location specified by a register
  - Add R4, (R1)
  - Second operand's value = Mem[Regs[R1]]

- **Displacement**: operand is in memory location specified by a register and some displacement
  - Add R4, 100(R1)
  - Second operand's value = Mem[Regs[R1]+100]

Addressing Modes

- **Indexed**: operand is in a memory location specified by a sum of the two register contents
  - Add R3, (R1+R2)
  - Second operand's value = Mem[Regs[R1]+Regs[R2]]

- **Direct or absolute**: operand is in a memory location specified by a constant
  - Add R1, (1001)
  - Second operand's value = Mem[1001]

Addressing Modes

- **Memory indirect**: operand is in a memory location which is specified by another memory location, stored in a register
  - Add R1, @ (R3)
  - Second operand's value = Mem[Mem[Regs[R3]]]

- **Autoincrement**: operand is in a memory location specified by a register; register value is postincremented by operand size
  - Add R1, (R2)+
  - Second operand's value = Mem[Regs[R2]]; Regs[R2]=Regs[R2]+d
Addressing Modes

- **Autodecrement**: operand is in a memory location specified by a register; register value is decremented by operand size
  - Add R1, -(R2)
  - Regs[R2] = Regs[R2] - d;
  - Second operand's value = Mem[Regs[R2]];
- **Scaled**: operand is in a memory location specified by two registers, a constant and a scale
  - Add R1, 100 (R2)[R3]
  - Second operand's value = Mem[100 + Regs[R2] + Regs[R3]*d]

Displacement Mode

- Add R4, 100(R1)
- How big a displacement should we support?

Immediate Mode

- Add R4, #3
- How big a constant should we support?
- Do we have to support constants for all instructions, or just for some?

Type And Size Of Operands

- Encoded in opcode
- Type of operand defines its size:
  - Character – 1 byte
  - Half word (short integer) – 2 bytes
  - Word (integer) – 4 bytes
  - Single precision FP and long integer – 4 bytes
  - Double precision FP – 8 bytes
  - Some architectures support packed decimal – two digits per byte

Control Flow Instructions

- Conditional branch
- Unconditional branch: jump
- Procedure call
- Procedure return

Addressing Modes For Control Flow Instructions

- Must specify the target address
  - Absolute
  - PC-relative – displacement is added to PC
  - Register-indirect
- PC-relative saves space as target is usually close to branch instruction and provides position independence
How To Specify Branch Condition?
- Condition code
- ALU operation sets special bits, get condition for free
- Constrain instruction ordering
- Condition register
  - Write 0 (false) or 1 (true) into a register after comparison
- Support only BZ and BNZ instructions
- Compare and branch
  - Compare operands (BLT, BGT, BEQ …) and branch
- Instruction may last long

Procedure Invocation Options
- Return address and some state must be saved
- Caller saving:
  - Calling procedure saves registers that it will need upon return
  - Must be used for globally accessed variables
- Callee saving:
  - Called procedure saves registers that it will overwrite

Operations in the Instruction Set
- Arithmetic: Add, multiply, subtract, divide
- Logical: And, or
- Control: branch, jump, procedure call and return
- System: OS call, virtual memory management
- FP operations: add, multiply, subtract, divide
- Decimal: add, multiply, convert
- String: move, compare, search
- Graphics: pixel and vertex operations

Encoding The Instruction Set
- Previous design decisions affect the size of the instruction
  - Size of the compiled program
  - Ease of decoding

Encoding The opcode Field
- Depends on whether every operation can be combined with every addressing mode
  - If it can separate address specifier is needed for each operand
  - If it can’t opcode can signify the addressing mode

Instruction Set Design Trade-offs
- More registers are better for compiler optimization
- More addressing modes bring faster operation
- More registers and addressing modes make instructions longer
- Shorter instructions and instructions with similar CPI are better for pipelining
**Instruction Formats**

**Variable**
- Works best if there are many operations and addressing modes
- All addressing modes with all instruction
- As few bits as possible to encode the program
- Decoding might be complicated

**Fixed**
- Works best if there is a small number of operations and addressing modes
- Larger programs
- Always same number bits to encode instructions
- Easy decoding

**Hybrid**
- Important for embedded applications
- Design hybrid version of instruction set with both 16-bit and 32-bit instructions
  - 16-bit instructions are simpler, support fewer operations and addressing modes
- Compressed code
  - Instruction cache contains full instructions
  - Memory contains compressed instructions
  - On cache miss, instruction is fetched and decompressed

**CISC vs. RISC**
- Complex Instruction Set Computer (CISC)
  - Instructions are highly specialized
  - Support for a variety of instructions, addressing modes, etc.
  - Different CPI and instruction size
- Reduced Instruction Set Computer (RISC)
  - Short, simple instructions, support for a few addressing modes
  - More complex instructions must be programmed
  - Same low CPI

**Role of Compilers**
- Compiler generates object code in machine language from the high-level language such as C
- Instruction set is compiler’s target
- In addition to generating the code, compiler optimizes the code to make it:
  - Shorter – 25% to 90%
  - Faster
  - Susceptible to pipelining
Compilation
- Compiler makes two to four passes through the code
  - In each pass it performs one of the optimizations
  - The optimizations are optional and may be skipped to achieve faster compilation
  - Passes are sequential
    - If compiler could go back and repeat steps it might discover better optimizations but this would increase time and complexity
- Compiler design goals:
  - Correctness
  - Speed of compilation

Front End
- Transforms high-level language into common intermediate representation
- When a new language becomes popular only front-end needs to be rewritten

High-Level Optimizations
- Transform the code to take advantage of parallelism and increase speed of execution:
  - Loop unrolling – expand body of the loop to encompass several iterations thus eliminating number of conditional branches
  - Procedure inlining – eliminates context switch
  - Prefetch insertion – prefetch array references in loops

Global Optimizations
- Global and local optimizations
  - Global common subexpression elimination – locates several expressions that compute same value and replaces the second with the temporary variable
  - Local optimization is done only within basic block
  - Copy propagation: if A=X replace all later references to A with X
- Register allocation
  - Allocate most accessed variables to registers
  - Since number of registers is limited, must find a strategy that does not result in too many transfers between the memory and the registers

Code Generator
- Takes advantage of design features of a specific architecture
  - Reorder instructions to improve pipeline performance
  - Replace multiplication with addition and shifts
Which Variables → Registers

- Program data allocation
  - Stack
    - Local scalar variables and activation records for procedures
    - Best for register allocation
  - Global area
    - Global variables and constants
    - Should be allocated to registers if accessed frequently
  - Heap
    - Dynamic objects accessed with pointers
    - Should not be allocated to registers
- Aliased variables should also not be allocated to registers

How Can Architecture Help?

- Provide regularity
  - Operations, data types and addressing modes should be orthogonal
- Provide primitives not solutions
  - Special features that match kernels or high-level languages are often unusable
- Simplify trade-offs among alternatives
  - Compilers strive to generate efficient code
  - Specify benefits and costs of each alternative
  - Make use of everything that is known at compile time

MIPS Instruction Set

We will not cover details in the lecture, but make sure you read this section!

since MIPS architecture is used in the material to follow