**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

**Reduced Code Size**
- 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

**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

**Compilation**

```
Front end per language
  \|-- High-level optimizations
     \|-- Global optimizer
        \|-- Code generator
```

**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
Pipelining
Appendix A

Pipelining – the Idea
- Similar to assembly line in a factory
- Divide instruction into smaller tasks
- Each task is performed on subset of resources
- Overlap the execution of multiple instructions by completing different tasks from different instructions in parallel
- Ideally this should lead to throughput of one instruction per clock cycle

Unpipelined Machine

Pipe-lined Machine

2 instructions completed in 6 cycles

2 instructions completed in 4 cycles

Pipelining Overview
- Each task is called pipe stage or pipe segment
- All stages must be able to proceed at the same time:
  - Stage duration is called the processor cycle
  - It is determined by the slowest stage
  - Stages should use disjoint resources if possible
- Goal of pipelining is to increase throughput — number of instructions completed per clock cycle
- In ideally balanced pipeline with n stages:
  \[ T_{\text{Instruction}} = \frac{T_{\text{Instruction, unpiped}}}{n} \]
  Speedup = n

Pipelining Overview
- Pipelining increases throughput, reduces the average execution time per instruction
- It does not reduce the time needed to execute each instruction
- Frequently this time is slightly increased due to overhead involved in passing between stages
Unpipelined MIPS

- Fetch instruction from Mem[PC]
- PC=PC+4

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
</table>

In parallel:
- Decode the instruction
- Read source registers, compare registers for possible branch
- Sign-extend the offset field if needed; compute possible branch target address by adding sign-extended offset to PC+4 (branch can now be completed)
- Note: we have two options for branch target calculation: in ID or in EX stage

If instruction is:
- Memory reference: add base register and offset to form memory address
- ALU: perform the operation; for immediate sign-extend the second operand

Write data into register file either for LOAD or for ALU instruction

If instruction is:
- Load: load data from memory
- Store: store data from register to memory