Compilers:
Code Generation

Dr Paris Yiapanis
room: John Dalton E151
email: p.yiapanis@mmu.ac.uk

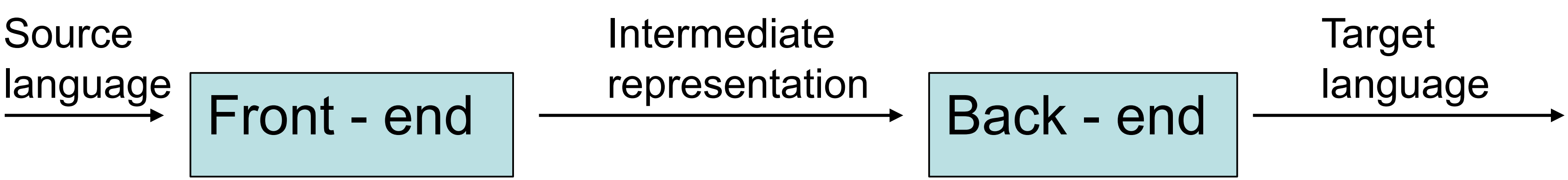
Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- ~~Machine-independent optimisation~~
- **Code generation**
- Hardware architectures
- Machine-dependent optimisation
- Review

Objectives

- To describe the principles in code generation
- To explain the roles of *instruction selection*, *register assignment* and *instruction scheduling*
- To develop a process for mapping from high-level language to machine code
- To explore patterns for common high-level constructs

High level compilation process



C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

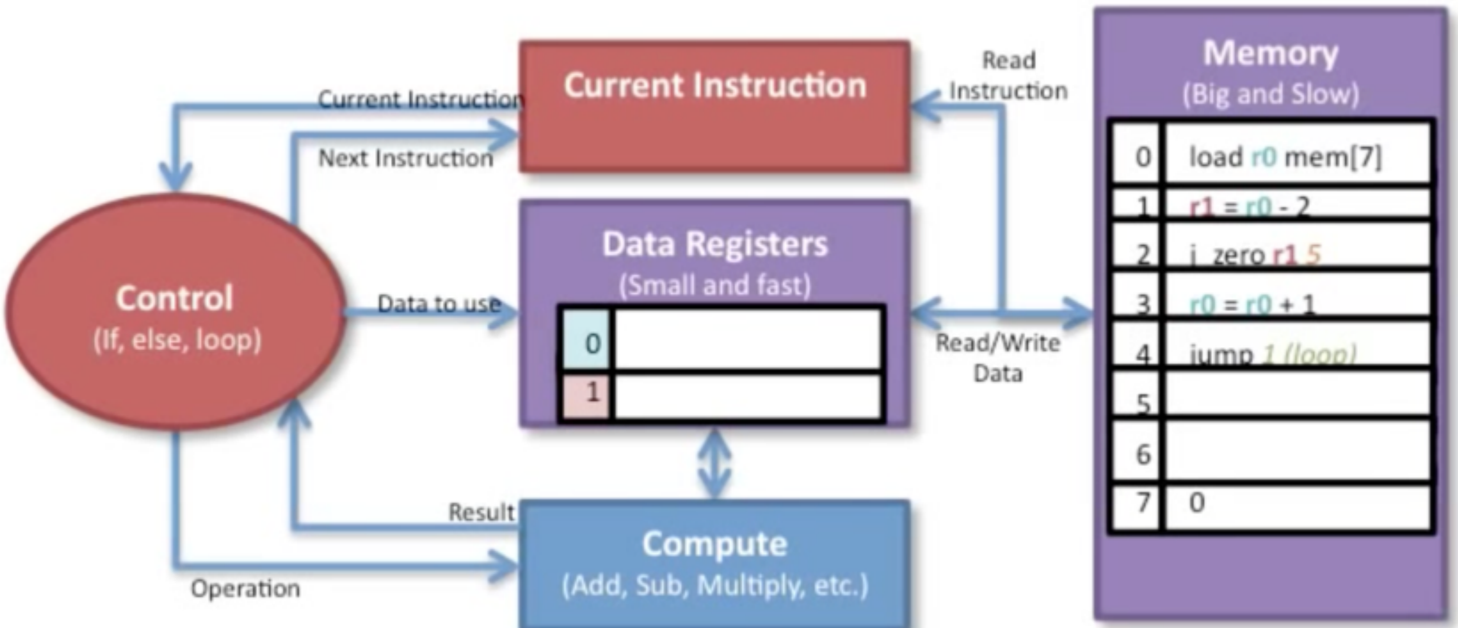
```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

OS Loader

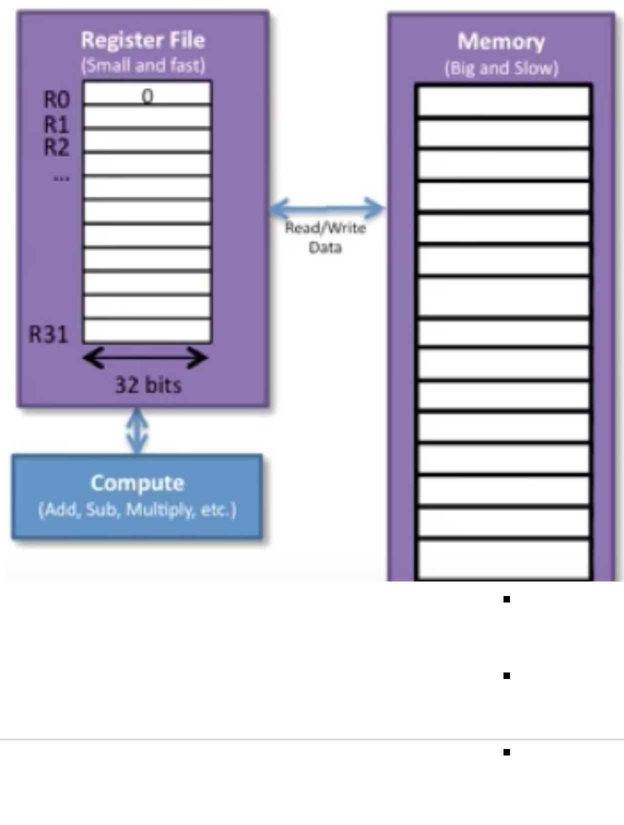
Memory

0	load r0 mem[7]
1	r1 = r0 - 2
2	j_zero r1 *5 (done)*
3	r0 = r0 + 1
4	jump *1 (loop)*
5	
6	
7	0

Simplified example instruction execution


Current Instruction
Next Instruction
Current Instruction
Read
Instruction
Memory
(Big and Slow)
0
load r0 mem[7]
1
r1 = r0 - 2
2
j_zero r1 5
3
r0 = r0 + 1
4
jump 1 (loop)
5
6
7
0
Control
(If, else, loop)
Data to use
Data Registers
(Small and fast)
0
1
Read/Write
Data
Result
Compute
(Add, Sub, Multiply, etc.)
Operation

Memory vs. Registers



Example MIPS instructions

- General 3-operand format
  - *op* dest, scr1, src2
- Addition
  - *add* $1, $2, $3
- Subtraction
  - *sub* $1, $2, $3

dest = src1 *op* src2

$1 = $2 + $3

$1 = $2 - $3

Types of instructions

- Data operations
  - Arithmetic (add, subtract, …)
  - Logical (and, or, not, xor)
- Data transfer
  - Load (*load* dest, src)
  - Store (*save* src, dest)
- Sequencing
  - Branch (conditional, e.g. <, >, ==)
  - Jump (unconditional, e.g. goto)

More complex example

- f = (g + h) - (i + j)

Assuming for simplicity that alphabet letters are memory locations of the form '0($0)'

```
load $1, g
load $2, h
add  $3, $1, $2

load $4, i
load $5, j
add  $6, $4, $5

sub $7, $3, $6
```

not the most efficient use of registers – could reuse

Code Generation

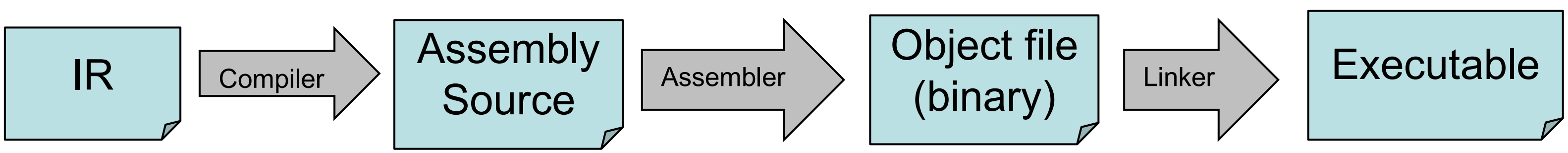
- Translates all the instructions in the intermediate representation into the target language
- Target program must preserve semantic meaning of source program
- Must make efficient usage of target machine resources

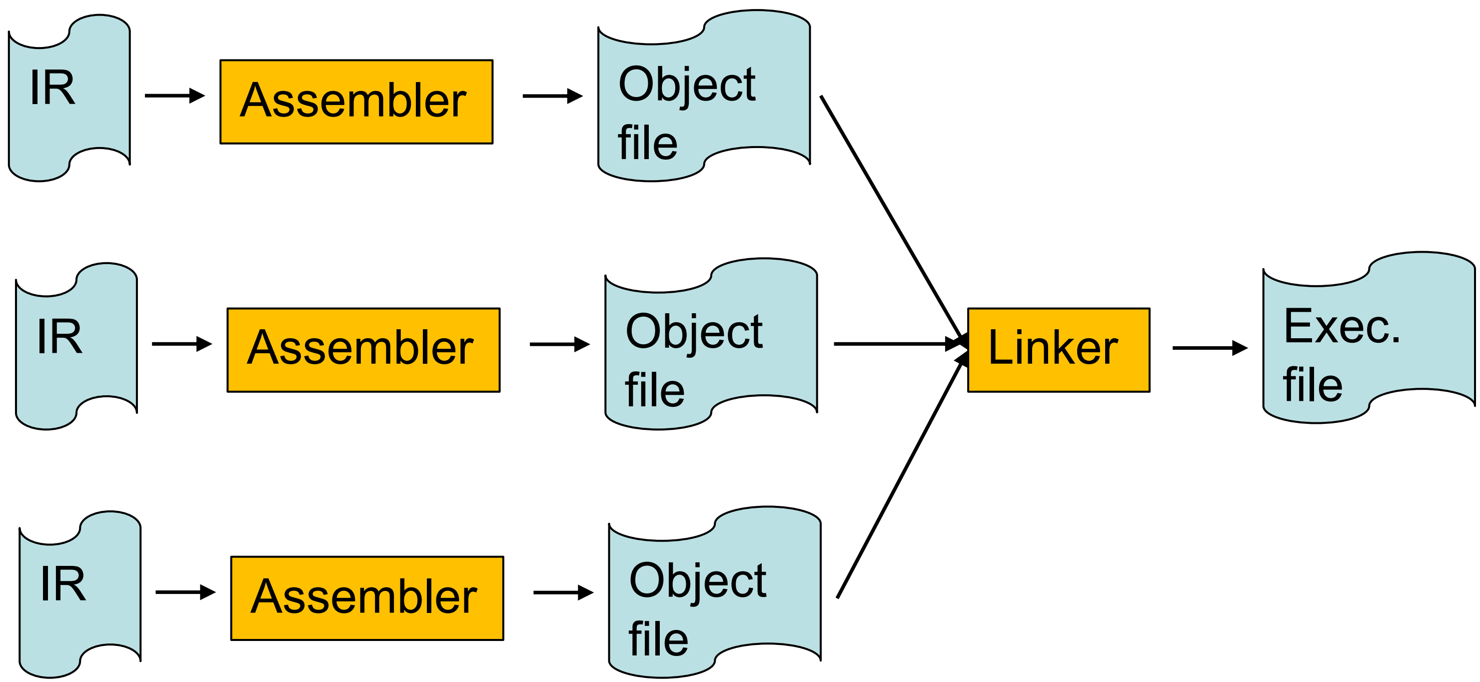
Code Generation example

```
a ← b + c
d ← a + e
```

```
load $0, b          //$0 = b
load $1, c          //$1 = c
add  $0, $0, $1     //$0 = $0 + $1
save $0, a          //a = $0
load $0, a          //$0 = a
load $1, e          //$1 = e
add  $0, $0, $1     //$0 = $0 + $1
save $0, d          //d = $0
```

Example compilation process (C/C++)




IR
Assembler
Object file
IR
Assembler
Object file
IR
Assembler
Object file
Linker
Exec. file

Design Issues

- Target Language
- Instruction selection
- Register allocation and assignment
- Instruction ordering

Design Issues: Target Language

- Instruction set architecture (RISC, CISC)
- Producing absolute machine-language program
- Producing relocatable machine-language program
- Producing assembly language programs

Assembly Language

- Advantages
  - Simplifies code generation due to use of symbolic instructions and symbolic names
    - Logical abstraction layer
  - Multiple Architectures can be described by a single assembly language
- Disadvantages
  - Additional process of assembling and linking
    - One extra step into compilation

Assembly Language

- Relocatable machine language (object modules)
  - All locations(addresses) represented by symbols
  - Mapped to memory addresses at link and load time
  - Flexibility of separate compilation
- Absolute machine language
  - Addresses are hard-coded
  - Simple and straightforward implementation inflexible – hard to reload generated code

Design Issues: Instruction Selection

- Choosing appropriate target-machine instructions to implement the IR statements
- The complexity of mapping IR program into code-sequence for target machine depends on:
  - Level of IR (high-level or low-level)
  - Nature of instruction set (data type support)
  - Desired quality of generated code (speed and size)

Example code generation to assembly*

*note: this is an abstract version of assembly language only used for demonstration purposes

Manchester Metropolitan University

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

C Program

```
while (i != 2) {
  i = i + 1;
}
```

Compiler

Assembly Code

```
load r0 mem[7]
loop:
  r1 = r0 - 2
  j_zero r1 done
  r0 = r0 + 1
  jump loop
done:
```

Instruction selection example

```
a ← b + c
d ← a + e
```

```
load $0, b          //$0 = b
load $1, c          //$1 = c
add  $0, $0, $1     //$0 = $0 + $1
save $0, a          //a = $0
load $0, a          //$0 = a
load $1, e          //$1 = e
add  $0, $0, $1     //$0 = $0 + $1
save $0, d          //d = $0
```

```
a ← b + c
d ← a + e
```

Load is redundant as result already in register $0

```
load $0, b          //$0 = b
load $1, c          //$1 = c
add  $0, $0, $1     //$0 = $0 + $1
save $0, a          //a = $0
load $0, a          //$0 = a
load $1, e          //$1 = e
add  $0, $0, $1     //$0 = $0 + $1
save $0, d          //d = $0
```

Design Issues: Registers

- **Register Allocation**: Selecting the set of variables that will reside in registers at each point in the program
- **Register Assignment**: Picking the specific register that a variable will reside in

Design Issues: Evaluation Order

- Selecting the order in which computations are performed
- Affects the efficiency of the target code
- Picking a best order is NP-complete
- Some orders require fewer registers than others

Back a step…

Machine Code is Pretty Basic!

- Many high-level language constructs don’t have a direct translation in machine code
- Need standard patterns for the common constructs
  - Loops (for, while, do…until)
  - Conditional statements (if…then...else, switch)

Loops

```
while (expr)
        do something
```

```
TEST:
        eval expr
        if cond == 0 goto END
        do something
        goto TEST
END:
```

```
        goto TEST
LOOP:
        do something
TEST:
        eval expr
        if cond != 0 goto LOOP
```

Conditionals

```
if (expr)
        do stmtsA
else
        do stmtsB
```

```
        eval expr
        if cond == 0 goto FALSE
        do stmtsA
        goto END
FALSE:
        do stmtsB
END:
```

Summary

- Purpose of code generation is…
  - (three closely-related issues)
- Basic operations in machine code…
  - Processor logic, memory management
- Translating high-level constructs to machine code
  - examples: loops and conditionals
- Relate translation to associated costs
  - (memory versus registers, optimisation)

Where we are…

- ~~Admin and overview~~
- ~~Lexical analysis~~
- ~~Parsing~~
- ~~Semantic analysis~~
- ~~Machine-independent optimisation~~
- ~~Code generation~~
- Hardware architectures
- Machine-dependent optimisation
- Review