Compilers

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

Today's Lecture

- Register Allocation

Compiler Phases: Front and Back Ends

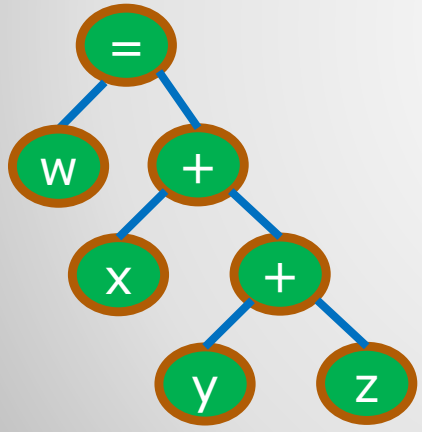
- The front-end phases are:
  - Scanning
  - Parsing
  - Semantic analysis
- The back-end phases are:
  - **Register Allocation**
  - Instruction Scheduling
  - Code generation

Instructions and Registers

- Registers are used to store variable values or temporary/intermediate results.
- Many assembly instructions need their input data to be stored in registers to execute.
  - An assembly add instruction may require its operands to be in a register.
  - For example: **add r1, r2**, r3
  - In the above line of code, the operands are in r1 and r2.
- Assembly instructions may store their result in a register.
  - For example: add r1, r2, **r3**
  - In the above line of code, the result is stored in r3.
- If two variables are being added they must be loaded into a register first. Here is the assembly code for x = y + z:
  - load r1, y
  - load r2, z
  - add r1, r2, r3
  - store r3, x

- Given the AST below, what is the assembly code for a virtual machine that has an unlimited number of registers.
- How many registers are needed in total?
- How many registers are needed to store intermediate results?

Abstract Syntax Tree
(Intermediate Representation



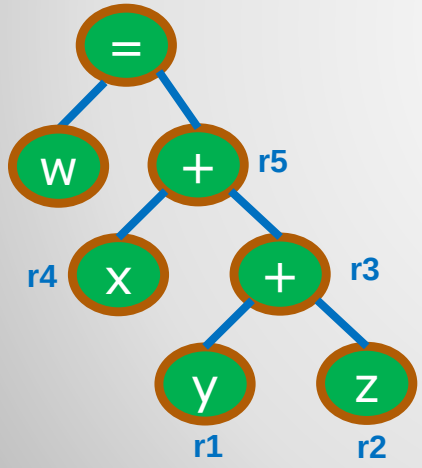
Assembly Code

???

Intermediate Values and Registers

- Three registers are needed for variables x, y, z.
- Two additional registers for results of add instructions.
  - The add instructions generate intermediate results that need to be stored in registers.
- Five total registers are needed.

Abstract Syntax Tree
(Intermediate Representation



Assembly Code

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

This code uses five registers in total.

Two registers are used for intermediate results (the results of the add instructions).

Intermediate Values and Registers

- A virtual machine has an unlimited number of registers.
- An intermediate representation of the code may be geared towards a virtual machine (assumes an infinite number of registers).
- Eventually, the code must be generated for a real machine which has a finite number of registers.

Virtual Registers vs Physical Registers

Register Allocation

- **Register allocation** - Maps virtual registers into physical registers.
- Virtual machine code assumes an infinite number of registers.
- Virtual machine code must be transformed into code that can run on a physical machine that has a finite number of registers.
- The register allocator is responsible for doing this mapping.

- Assume virtual registers r1, r2 and physical registers p1,p2.
- Here is a mapping of virtual to physical registers:

Mapping

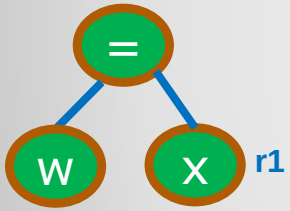
r1 → p1
r2 → p2

Live Variable Information

- **Live variable** – A variable is live if it will be used by code that comes later in the program.
- The register allocator needs to keep track of which values in registers are still needed (which values in registers are still live).
- The register allocator does not need to keep values in registers unless they are live (because those values are no longer needed, code that follows will not use those values).

- Map the virtual registers to physical registers.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
load r1, x
store r1, w
```

Mapping

r1 → p1

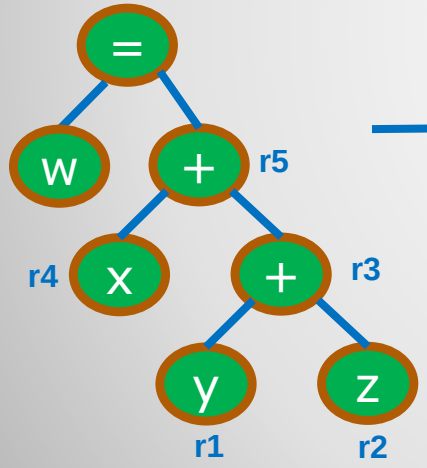
Physical Assembly

```
var w
var x
load p1, x
store p1, w
```

Map Virtual to Physical Registers

- Assume a physical machine with five registers. Call them p1, p2, p3, p4, p5.
- Map the virtual registers to physical registers.
- What is the physical assembly code?
- Hint: Same number of virtual and physical registers.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

???

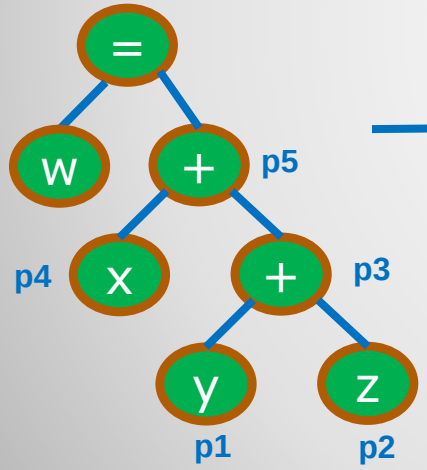
Physical Assembly

???

Map Virtual to Physical Registers 1

- Assume a physical machine with five registers. Call them p1, p2, p3, p4, p5.
- There are enough physical registers to match the virtual registers.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

r1 → p1
r2 → p2
r3 → p3
r4 → p4
r5 → p5

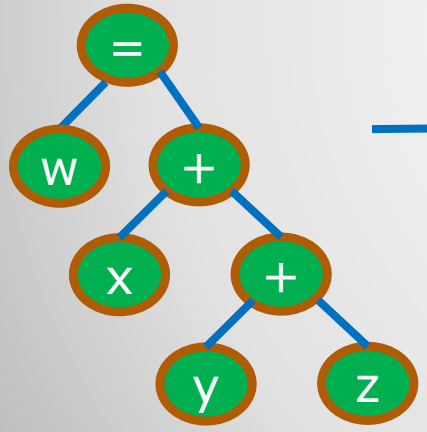
Physical Assembly

```
var w
var x
var y
var z
load p1, y
load p2, z
add p1, p2, p3
load p4, x
add p3, p4, p5
store p5, w
```

Map Virtual to Physical Registers 1

- Assume a physical machine with three registers. Call them p1, p2, p3.
- Map the virtual registers to physical registers.
- What is the physical assembly code?
- Hint: Think about which registers are live.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

???

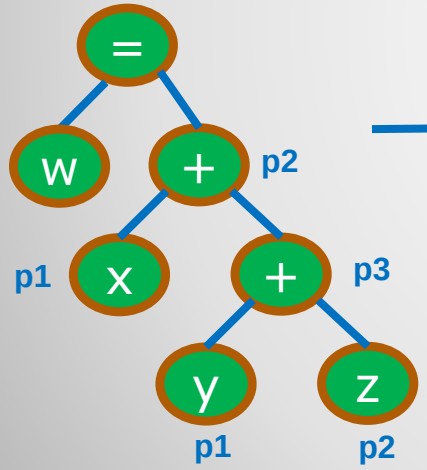
Physical Assembly

???

Map Virtual to Physical Registers 2

- Assume a physical machine with <u>three</u> registers. Call them p1, p2, p3.
- This code reuses p1 and p2.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

r1 → p1
r2 → p2
r3 → p3
r4 → p1
r5 → p2

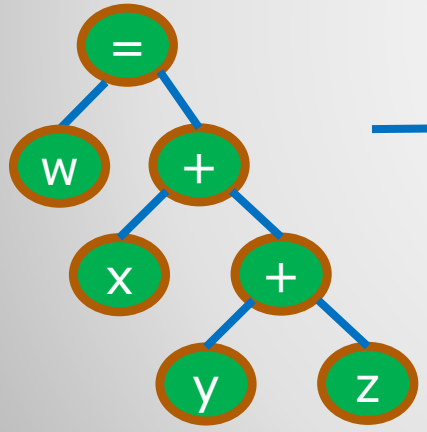
Physical Assembly

```
var w
var x
var y
var z
load p1, y
load p2, z
add p1, p2, p3
load p1, x
add p1, p3, p2
store p2, w
```

Map Virtual to Physical Registers 2

- Assume a physical machine with two registers. Call them p1, p2.
- Map the virtual registers to physical registers.
- What is the physical assembly code?
- Hint: You can reuse a register inside of a single instruction.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

???

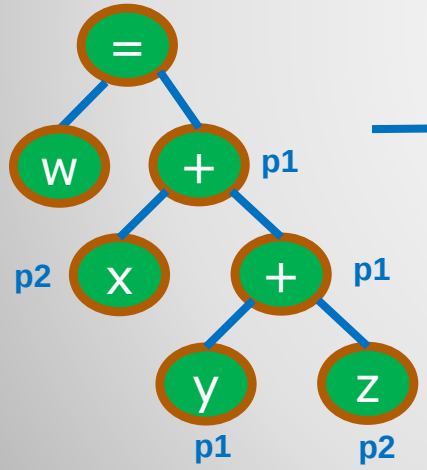
Physical Assembly

???

Map Virtual to Physical Registers 3

- Assume a physical machine with <u>two</u> registers. Call them p1, p2.
- This code reuses p1 more than once.

Abstract Syntax Tree (Intermediate Representation



Virtual Assembly

```
var w
var x
var y
var z
load r1, y
load r2, z
add r1, r2, r3
load r4, x
add r3, r4, r5
store r5, w
```

Mapping

r1 → p1
r2 → p2
r3 → p1
r4 → p2
r5 → p1

Physical Assembly

```
var w
var x
var y
var z
load p1, y
load p2, z
add p1, p2, p1
load p2, x
add p1, p2, p1
store p1, w
```

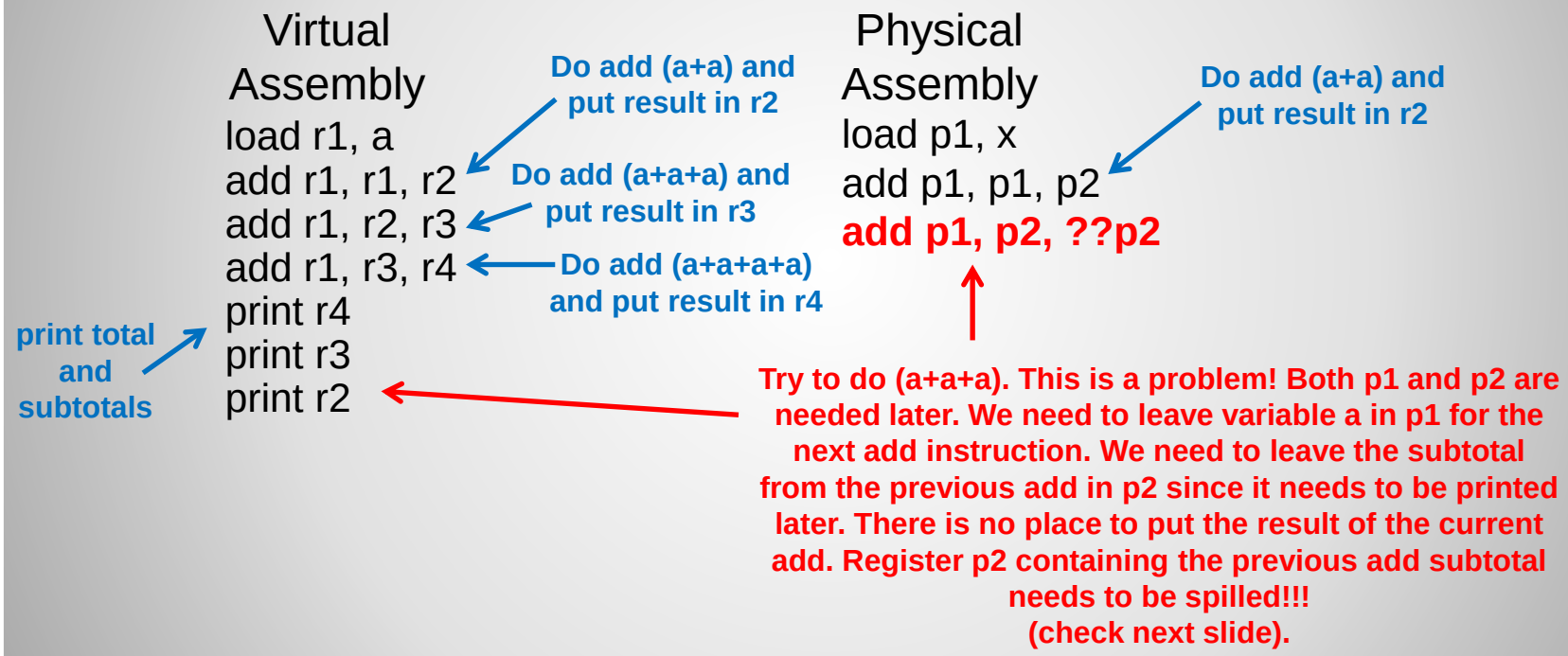
Map Virtual to Physical Registers 3

Spilling

- If there are not enough physical registers to hold all required data, then some data must be stored in RAM.
- Storing data in RAM because there are no free registers is called spilling.
- The register allocator tries to minimize spilling.
- The reason spilling must be minimized is because load and store instructions are time consuming operations and will slow down the program.

Spilling Example

- Spilling example.
- Calculate a+a+a+a and print subtotals and total.
- Assume there are only two physical registers:



Spilling Example

- Calculate a+a+a+a and print subtotals and total.
- Spilling code is added below to save subtotals to RAM. Need to generate temp variable names for spilled data.
- Assume there are only <u>two</u> physical registers:

Virtual Assembly	Mapping
load r1, a	r1 → p1
add r1, r1, r2	r2 → p2
add r1, r2, r3	r3 → p2
add r1, r3, r4	r4 → p2
print r4	
print r3	
print r2	

Registers r2, r3, and r4 all map to the same physical register

Physical Assembly

```
load p1, x
add p1, p1, p2
store p2, tempVar1
add p1, p2, p2
store p2, tempVar2
add p1, p2, p2
print p2
load p2, tempVar2
print p2
load p2, tempVar1
print p2
```

Do add (a+a) and put result in r2

Spill the first subtotal to RAM (a+a)

Do next add, overwrites subtotal in p2

Spill the second subtotal to RAM (a+a+a)

Do next add, overwrites subtotal in p2 (final total)

Load and print first subtotal

Load and print second subtotal

Register Allocation Algorithms

- Easy Version.
  - Store results of all calculations immediately to RAM.
  - This approach requires a small number of physical registers (two should be enough most of the time).
  - It is very slow though.
- Challenging Version.
  - Keep track of live variables.
  - Assign live variables to registers (nontrivial). Can use a graph coloring algorithm to assign variables to registers (colors represent registers).
  - Generated code runs much faster than the easy version code.
  - More complex to implement.

- **End of Slides**

End of Slides