Compilers and computer architecture

14. Code generation for register machines

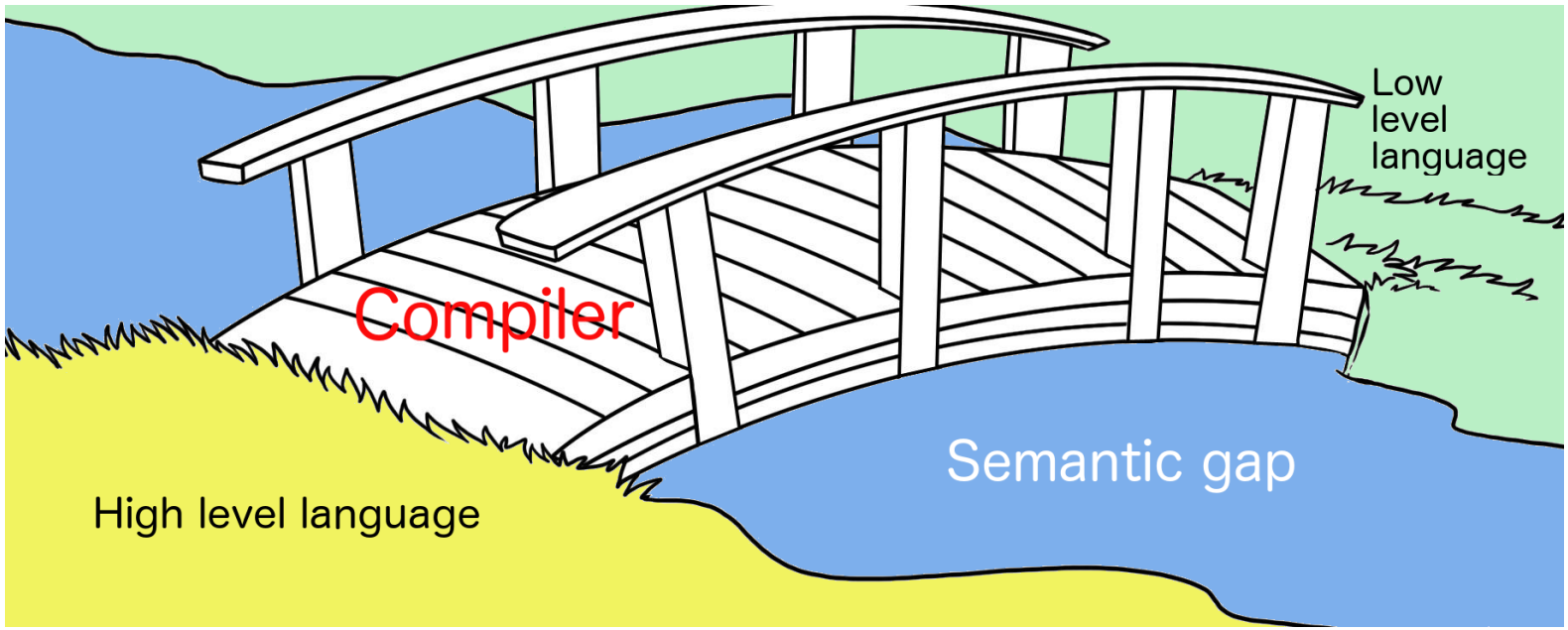
Hsi-Ming Ho

(slides by Martin Berger)

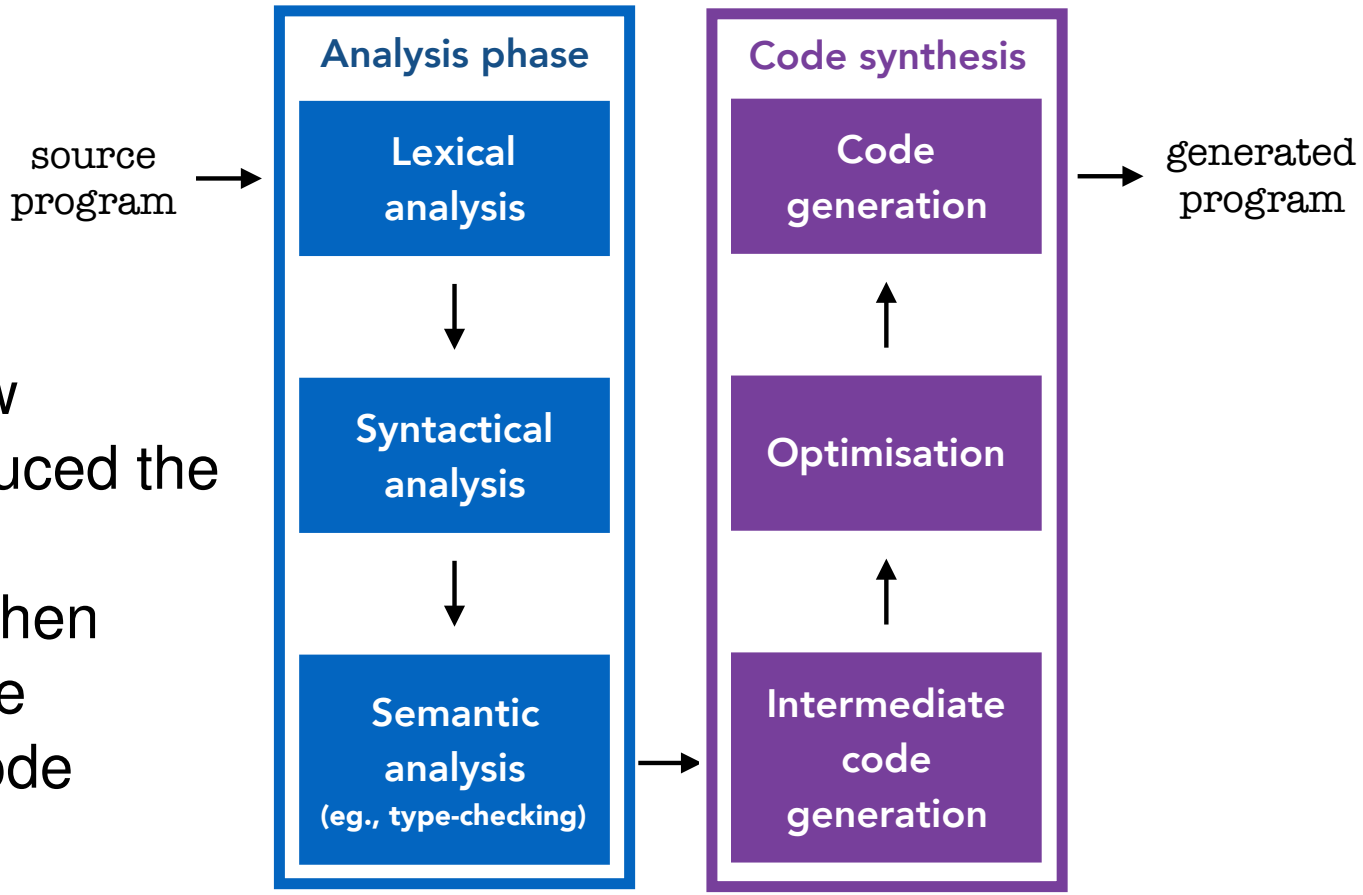
Email: **hsi-ming.ho@sussex.ac.uk**
Office hours: Online, by email appointment

Autumn 2022

Recall the function of compilers

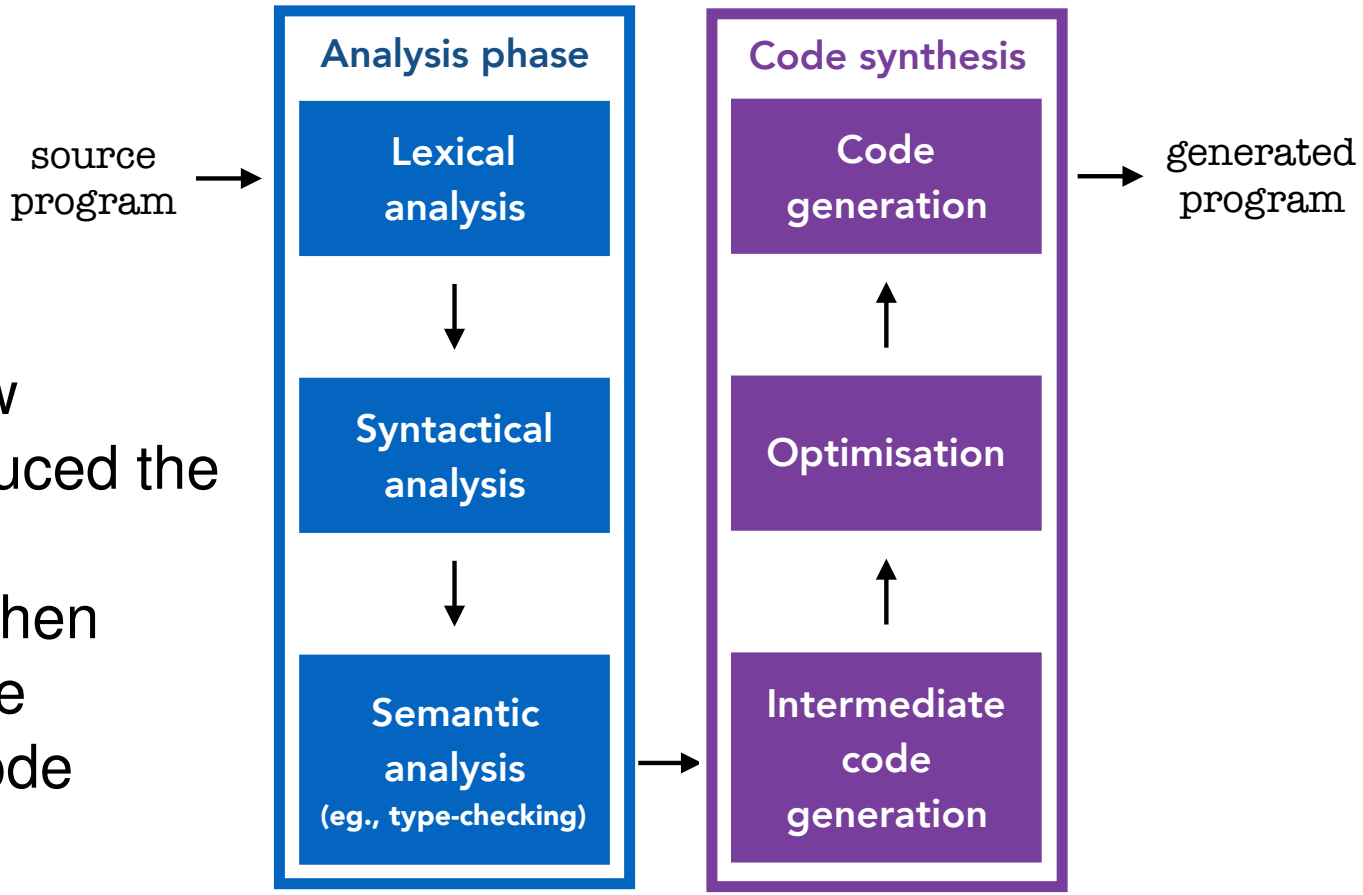


Where we are



In the previous few lectures we introduced the stack machine architecture, and then discussed a simple syntax-directed code generator for it.

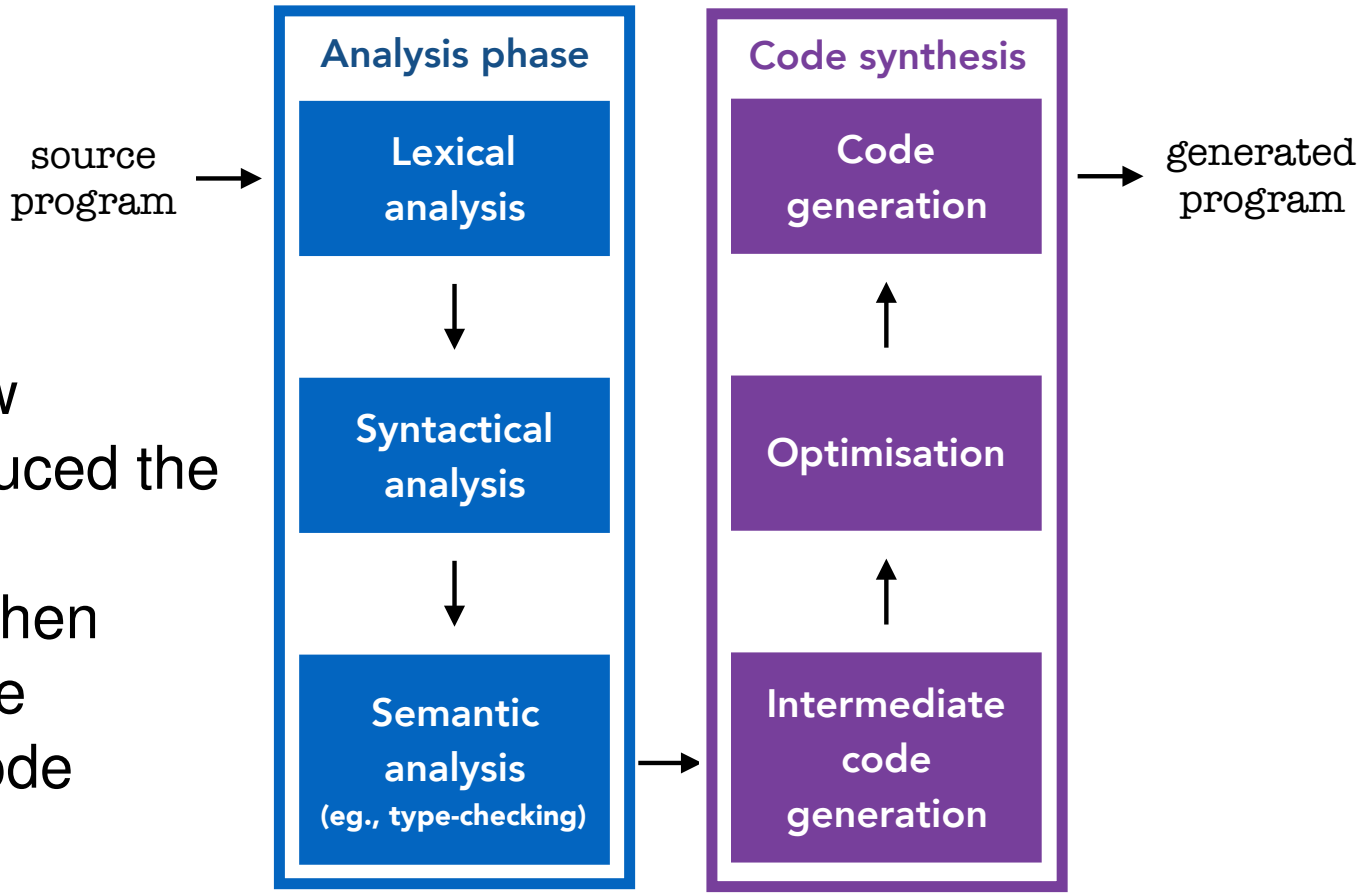
Where we are



In the previous few lectures we introduced the stack machine architecture, and then discussed a simple syntax-directed code generator for it.

In this lecture we continue on code generation, but for *register machines*, a faster architecture.

Where we are

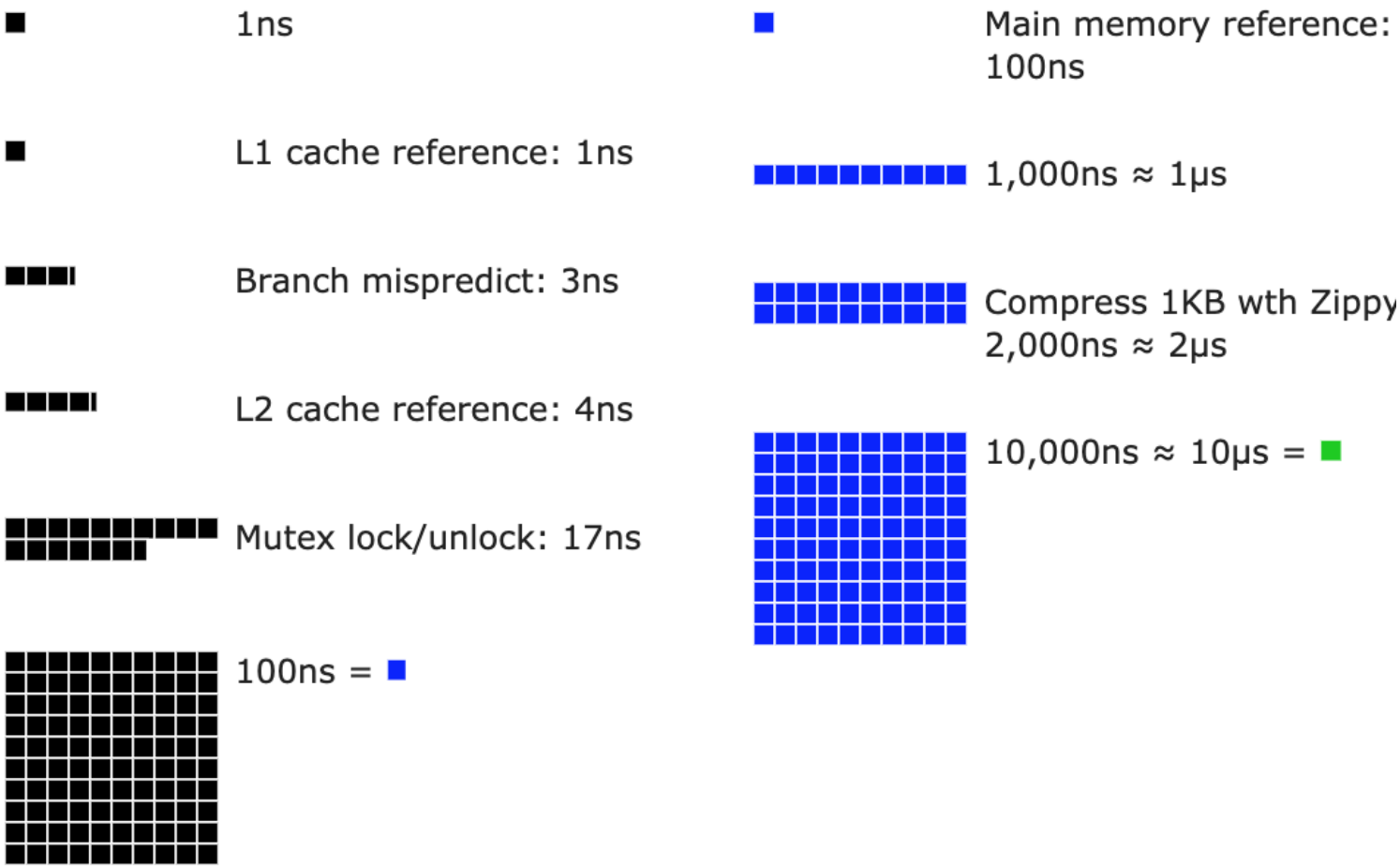


In the previous few lectures we introduced the stack machine architecture, and then discussed a simple syntax-directed code generator for it.

In this lecture we continue on code generation, but for *register machines*, a faster architecture.

Question: How an architecture can be ‘faster’ than another?

Memory wall



From: `https://people.eecs.berkeley.edu/~rcs/research/interactive_latency.html`

Register machines

The problem with the stack machines is that memory access (on modern CPUs) is **very slow** in comparion with CPU operations, approx. 20-100 times slower.

Register machines

The problem with the stack machines is that memory access (on modern CPUs) is **very slow** in comparion with CPU operations, approx. 20-100 times slower.

The stack machine forces us constantly to access memory, even for the simplest operations. It would be nice if the CPU let us store, access and manipulate data more directly, rather than only work on/through the top elements of the stack.

Register machines

The problem with the stack machines is that memory access (on modern CPUs) is **very slow** in comparion with CPU operations, approx. 20-100 times slower.

The stack machine forces us constantly to access memory, even for the simplest operations. It would be nice if the CPU let us store, access and manipulate data more directly, rather than only work on/through the top elements of the stack.

Registers are **fast** (in comparison with memory), temporary, addressable storage in the CPU. In general, (physical) register machines are faster than (physical) stack machines!

Register machines

The problem with the stack machines is that memory access (on modern CPUs) is **very slow** in comparion with CPU operations, approx. 20-100 times slower.

The stack machine forces us constantly to access memory, even for the simplest operations. It would be nice if the CPU let us store, access and manipulate data more directly, rather than only work on/through the top elements of the stack.

Registers are **fast** (in comparison with memory), temporary, addressable storage in the CPU. In general, (physical) register machines are faster than (physical) stack machines!

But compilation for register machines is more complicated than compilation for stack machines. Why?

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

For small expressions, we can fit all the relevant arguments into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on whether its value will be used in the future (*cf.* liveness analysis).

Let's just be pessimistic and revert back to memory when run out of registers. But then a compiler must still be able to do the following things:

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

For small expressions, we can fit all the relevant arguments into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on whether its value will be used in the future (*cf.* liveness analysis).

Let's just be pessimistic and revert back to memory when run out of registers. But then a compiler must still be able to do the following things:

- Generate code that puts all arguments in registers.

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

For small expressions, we can fit all the relevant arguments into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on whether its value will be used in the future (*cf.* liveness analysis).

Let's just be pessimistic and revert back to memory when run out of registers. But then a compiler must still be able to do the following things:

- Generate code that puts all arguments in registers.
- Generate code that puts most (all?) arguments in the memory.

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

For small expressions, we can fit all the relevant arguments into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on whether its value will be used in the future (*cf.* liveness analysis).

Let's just be pessimistic and revert back to memory when run out of registers. But then a compiler must still be able to do the following things:

- Generate code that puts all arguments in registers.
- Generate code that puts most (all?) arguments in the memory.
- Detect which of the above is the case, and be able seamlessly to switch between the two.

Compilation for register machines

Each CPU has only a small finite number of registers (e.g. 16, 32, 128). This can be a problem. Why?

For small expressions, we can fit all the relevant arguments into the registers, but for the execution of larger expressions this is no longer the case. Moreover, what registers are available at each point in the computation depends on whether its value will be used in the future (*cf.* liveness analysis).

Let's just be pessimistic and revert back to memory when run out of registers. But then a compiler must still be able to do the following things:

- Generate code that puts all arguments in registers.
- Generate code that puts most (all?) arguments in the memory.
- Detect which of the above is the case, and be able seamlessly to switch between the two.

All of this makes compilation more difficult.

A simple register machine

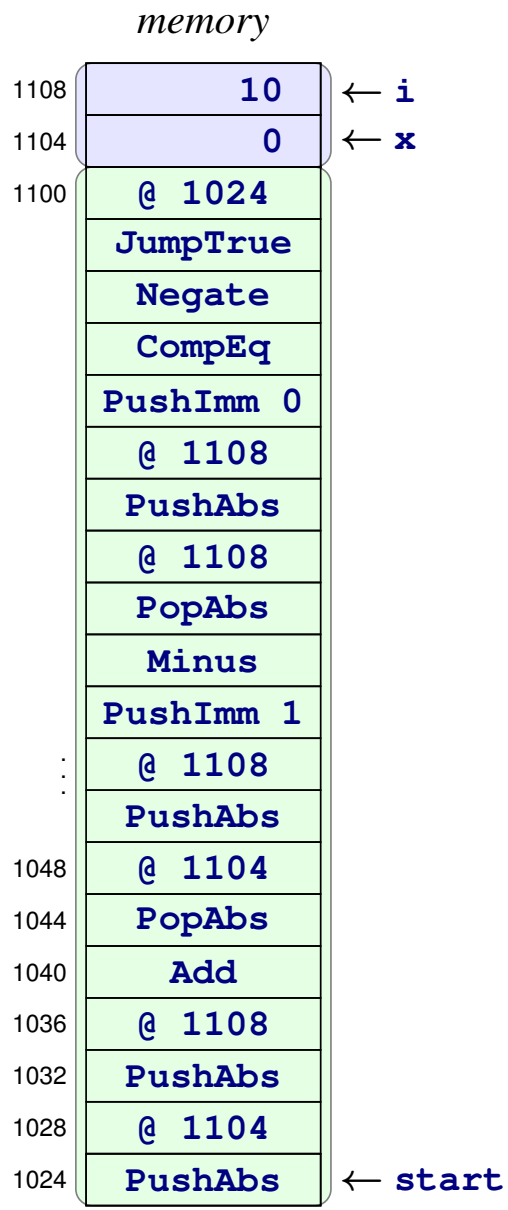
```
start:
    Load        R0, x
    Load        R1, i
    Add         R2, R0, R1
    Store       x
    Load        R3, i
    LoadImm     R4, 1
    Minus       R3, R4
    Store       R3, i
    Load        R5, i
    LoadImm     R6, 0
    CompEq      R5, R6
    JumpFalse   R5, start
```

A simple register machine

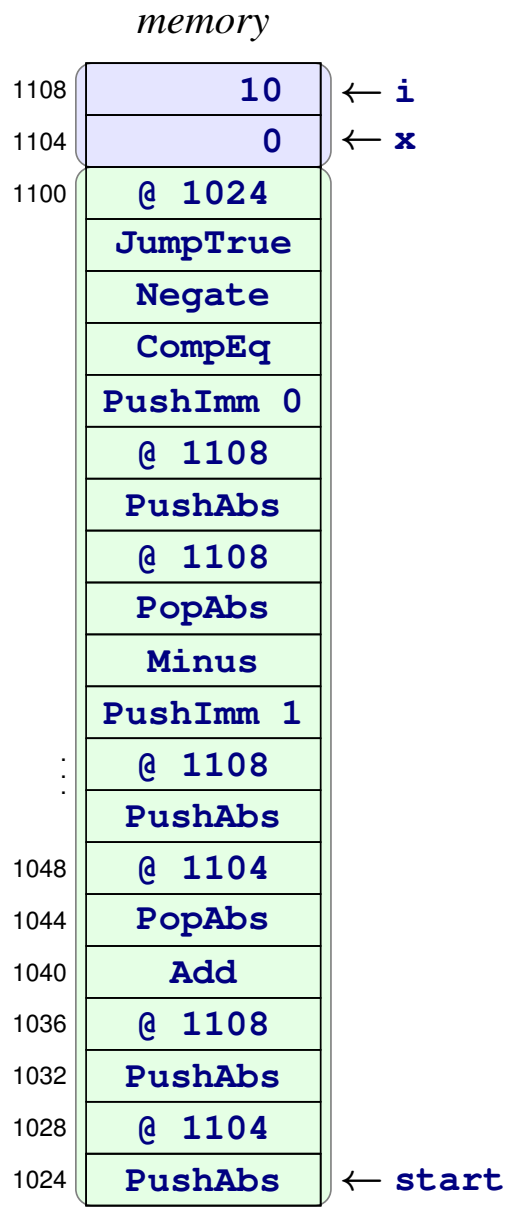
```
start:
    Load        R0, x
    Load        R1, i
    Add         R2, R0, R1
    Store       x
    Load        R3, i
    LoadImm     R4, 1
    Minus       R3, R4
    Store       R3, i
    Load        R5, i
    LoadImm     R6, 0
    CompEq      R5, R6
    JumpFalse   R5, start
```

Quite similar to the stack machine, but we have additional registers.
Important: operations like add, multiply operate on registers, instead of on the top of the stack.

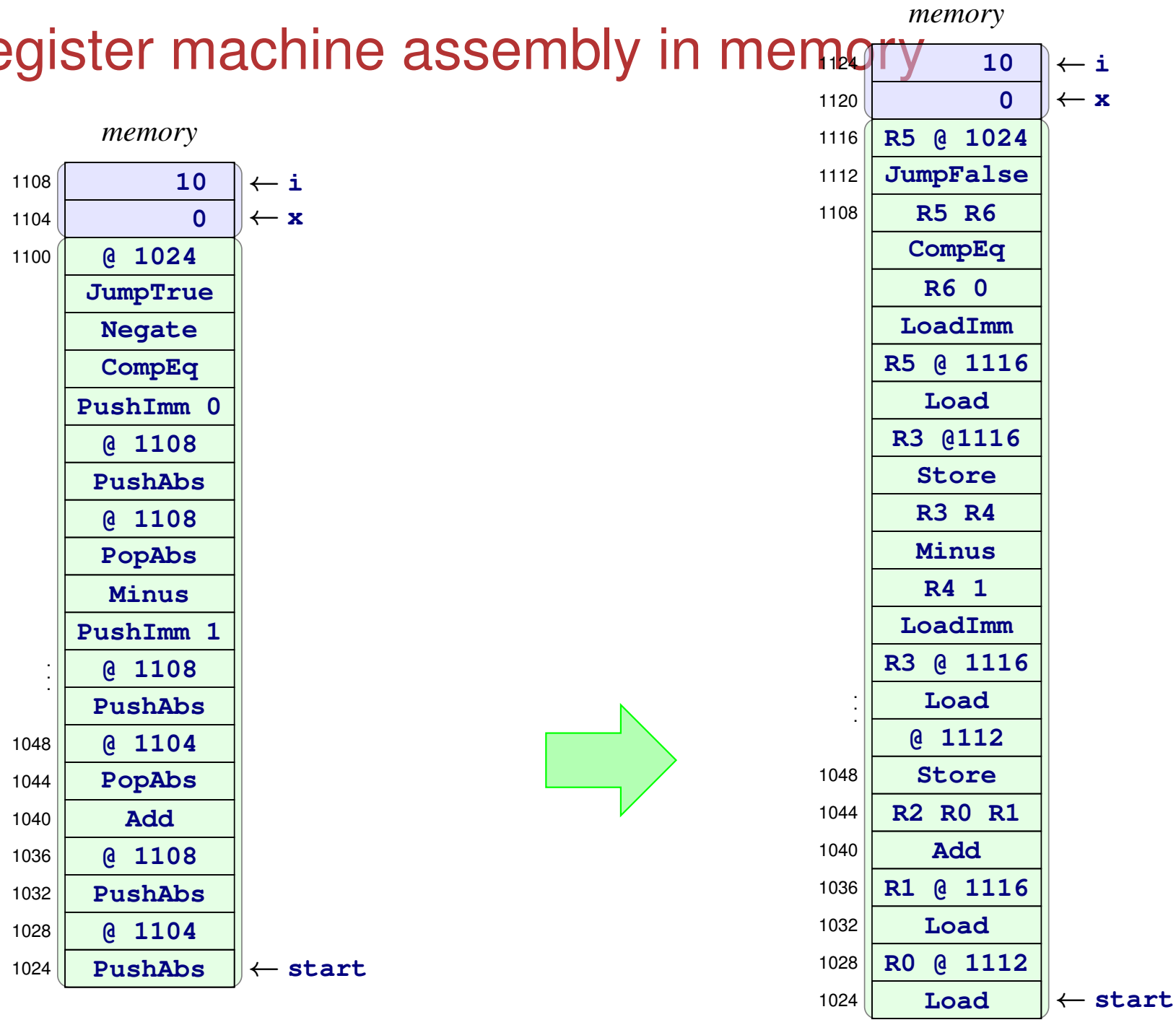
Register machine assembly in memory



Register machine assembly in memory



Register machine assembly in memory



Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating **efficient** code for register machines (which also have stacks!), we split the problem into three simpler sub-problems.

Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating **efficient** code for register machines (which also have stacks!), we split the problem into three simpler sub-problems.

- Generate code assuming an unlimited supply of registers.

Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating **efficient** code for register machines (which also have stacks!), we split the problem into three simpler sub-problems.

- Generate code assuming an unlimited supply of registers.
- Modify the generator to evaluate expressions in the order which minimises the number of registers needed.

Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating **efficient** code for register machines (which also have stacks!), we split the problem into three simpler sub-problems.

- Generate code assuming an unlimited supply of registers.
- Modify the generator to evaluate expressions in the order which minimises the number of registers needed.
- Invent schemes to handle cases where we run out of registers (what do we do with new expressions then?)

Dealing with registers by 'divide-and-conquer'

In order to explain the difficult problem of generating **efficient** code for register machines (which also have stacks!), we split the problem into three simpler sub-problems.

- Generate code assuming an unlimited supply of registers.
- Modify the generator to evaluate expressions in the order which minimises the number of registers needed.
- Invent schemes to handle cases where we run out of registers (what do we do with new expressions then?)

Let's start by looking at register machines with an **unlimited** number of registers.

Commands for register machines

Commands for register machines

We assume an **unlimited** supply of registers R0, R1, R2, ... ranged over by r, r'. We call these *general-purpose registers* (as distinct from PC, SP).

Commands for register machines

We assume an **unlimited** supply of registers R0, R1, R2, ... ranged over by r, r'. We call these *general-purpose registers* (as distinct from PC, SP).

`Nop`	Does nothing
`Pop r`	removes the top of the stack and stores it in register r
`Push r`	Pushes the content of the register r on stack
`Load r x`	Loads the content of memory location x into register r
`LoadImm r n`	Loads integer n into register r
`Store r x`	Stores the content of register r in memory location x
`CompGt r r'`	Compares the content of register r with the content of register r'. Stores 1 in r if former is larger than latter, otherwise stores 0

Commands for register machines

`CompEq r r'`	Compares the content of register r with the content of register r. Stores 1 in r if both are equal, otherwise stores 0
`Jump l`	Jumps to l
`JumpTrue r l`	Jumps to address/label l if the content of register r is not 0
`JumpFalse r l`	Jumps to address/label l if the content of register r is 0
`Plus r r'`	Adds the content of r and r', put the result in r
...	Remaining arithmetic operations are similar

Commands for register machines

`CompEq r r'`	Compares the content of register r with the content of register r. Stores 1 in r if both are equal, otherwise stores 0
`Jump l`	Jumps to l
`JumpTrue r l`	Jumps to address/label l if the content of register r is not 0
`JumpFalse r l`	Jumps to address/label l if the content of register r is 0
`Plus r r'`	Adds the content of r and r', put the result in r
...	Remaining arithmetic operations are similar

Some commands have arguments (called operands). They take two (if the command has one operand) or three units of storage, the others only one. These operands need to be specified in the opcode, unlike with the stack machine. (Why?)

Commands for register machines

`CompEq r r'`	Compares the content of register r with the content of register r. Stores 1 in r if both are equal, otherwise stores 0
`Jump l`	Jumps to l
`JumpTrue r l`	Jumps to address/label l if the content of register r is not 0
`JumpFalse r l`	Jumps to address/label l if the content of register r is 0
`Plus r r'`	Adds the content of r and r', put the result in r
...	Remaining arithmetic operations are similar

Some commands have arguments (called operands). They take two (if the command has one operand) or three units of storage, the others only one. These operands need to be specified in the opcode, unlike with the stack machine. (Why?)

Question: Why do we bother with stack operations at all?

Source language

Our source language is the same as before: a really simple imperative language.

$$\begin{array}{rcl} M & ::= & M; M \mid \texttt{for}\ x\ \texttt{=}\ E\ \texttt{to}\ E\ \texttt{\{}\ M\ \texttt{\}} \mid x := E \\ E & ::= & n \mid x \mid E + E \mid E - E \mid E * E \mid E / E \mid -E \end{array}$$

Note that `for` introduces a new binding scope, e.g.,

```
for i = 1 to 1 {
    for j = 1 to 5 {
        i := i * j;
    }
    // i is the value of 5! here
}
```

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
            : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
            : List [ Instruction ]
```

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
            : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
            : List [ Instruction ]
```

Convention 1:

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
            : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
            : List [ Instruction ]
```

Convention 1: The result of evaluating the expression is returned in register **`target`**.

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
            : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
            : List [ Instruction ]
```

Convention 1: The result of evaluating the expression is returned in register **`target`**.

Convention 2:

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
           : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
           : List [ Instruction ]
```

Convention 1: The result of evaluating the expression is returned in register `target`.

Convention 2: The code generated by `codegenExpr( e, i )` can use registers Ri, Ri+1, ... **upwards**, but must leave the other registers R0 ... Ri-1 **unchanged**!

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
           : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
           : List [ Instruction ]
```

Convention 1: The result of evaluating the expression is returned in register `target`.

Convention 2: The code generated by `codegenExpr( e, i )` can use registers Ri, Ri+1, ... **upwards**, but must leave the other registers R0 ... Ri-1 **unchanged**!

One way of thinking about this `target` is that it is used to track where the stack pointer would point in a stack machine.

Code generation for register machines

```
def codegen ( s : AST, target :  Register )
            : List [ Instruction ]

def codegenExpr ( exp : Expr, target :  Register )
            : List [ Instruction ]
```

Convention 1: The result of evaluating the expression is returned in register **`target`**.

Convention 2: The code generated by **`codegenExpr( e, i )`** can use registers Ri, Ri+1, ... **upwards**, but must leave the other registers R0 ... Ri-1 **unchanged**!

One way of thinking about this **`target`** is that it is used to track where the stack pointer would point in a stack machine.

Similar conventions for **`codegen`** for statements.

Code generation for constants

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
        ...
      Const ( n ) then
           List ( I_LoadImm ( target, n ) ) } }
```

Code generation for variables

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
          ...
    Ident ( x ) then
         List ( I_Load ( target, x ) )
```

Code generation for binary expressions

```
def codegenExpr ( exp : Expr, target : Register ) = {
    if exp is of shape
           ...
    Binop ( lhs, op, rhs ) then {
        codegenExpr ( rhs, target ) ++
        codegenExpr ( lhs, target+1 ) ++
        codegenBinop ( op, target, target+1 ) }
```

where

```
def codegenBinop ( op : Op, r1 : Register,
                            r2 : Register ) = {
    if op is of shape
     Plus then List ( I_Plus ( r1, r2 ) )
     Minus then List ( I_Minus ( r1, r2 ) )
     Times then List ( I_Times ( r1, r2 ) )
     Divide then List ( I_Divide ( r1, r2 ) ) } }
```

Code generation for binary expressions

Note that the call **`codegenExpr(lhs, target+1)`** in

```
Binop ( lhs, op, rhs ) then {
    codegenExpr ( rhs, target ) ++
    codegenExpr ( lhs, target+1 ) ++
    codegenBinop ( op, target, target+1 ) }
```

leaves the result of the first call **`codegenExpr(rhs, target)`** in the register **`target`** unchanged by our assumptions that **`codegenExpr`** never modifies registers below its second argument.

Code generation for binary expressions

Note that the call **`codegenExpr(lhs, target+1)`** in

```
Binop ( lhs, op, rhs ) then {
    codegenExpr ( rhs, target ) ++
    codegenExpr ( lhs, target+1 ) ++
    codegenBinop ( op, target, target+1 ) }
```

leaves the result of the first call **`codegenExpr(rhs, target)`** in the register **`target`** unchanged by our assumptions that **`codegenExpr`** never modifies registers below its second argument.

Please convince yourself that each clause of **`codegenExpr`** really implements this guarantee!

Example (x*3)+4

Compiling the expression (x*3)+4 (to target register r17) gives:

Example (x*3)+4

Compiling the expression (x*3)+4 (to target register r17) gives:

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

How can this be improved (1)?

Let's use commutativity of addition (a+b = b+a) and compile 4+(x*3)! When we compile it, we obtain:

How can this be improved (1)?

Let's use commutativity of addition (a+b = b+a) and compile 4+(x*3)! When we compile it, we obtain:

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

How is this better?

Side by side

Compilation of (x*3)+4

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

Compilation of 4+(x*3)

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

Side by side

Compilation of (x*3)+4

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

Compilation of 4+(x*3)

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

The translation on the left uses 3 registers, while the right only two. We are currently assuming an unbounded number of registers, so who cares ...

Side by side

Compilation of (x*3)+4

```
LoadImm r17 4
LoadImm r18 3
Load r19 x
Times r18 r19
Plus r17 r18
```

Compilation of 4+(x*3)

```
LoadImm r17 3
Load r18 x
Times r17 r18
LoadImm r18 4
Plus r17 r18
```

The translation on the left uses 3 registers, while the right only two. We are currently assuming an unbounded number of registers, so who cares ... For realistic CPUs the number of registers is small, so smart translation strategies that save registers are better.

Translation of statements

Similar to stack machine, except that arguments and results of expressions are held in registers. We'll see this in detail later.

Translation of statements

Similar to stack machine, except that arguments and results of expressions are held in registers. We'll see this in detail later.

Question: Does the `codegenStatement` method need to be passed a target register?

Translation of statements

Similar to stack machine, except that arguments and results of expressions are held in registers. We'll see this in detail later.

Question: Does the `codegenStatement` method need to be passed a target register?

Answer: Yes, because statements may contain expressions, e.g. `x := x*y+3`.

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting?

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting? We can combine two strategies:

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting? We can combine two strategies:

- While > 1 free registers remain, be 'greedy': use the register machine strategy discussed above for compilation.

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting? We can combine two strategies:

- While > 1 free registers remain, be 'greedy': use the register machine strategy discussed above for compilation.
- When the limit is reached (ie. when there is one register left), revert to the accumulator strategy, using the last register as the accumulator.

Bounded register numbers

It's easy to compile to register machine code, when the number of registers is unlimited. Now we look at compilation to register machines with a fixed number of registers.

Let's go to the other extreme: just one register called **accumulator**. Operations take one of their arguments from the accumulator, and store the result in the accumulator. Additional arguments are taken from the top of the stack.

Why is this interesting? We can combine two strategies:

- While > 1 free registers remain, be 'greedy': use the register machine strategy discussed above for compilation.
- When the limit is reached (ie. when there is one register left), revert to the accumulator strategy, using the last register as the accumulator.

The effect is that most expressions get the full benefit of registers, while unusually large expressions are handled correctly.

Two-phase strategy translation

Now we do what we promised earlier.

Two-phase strategy translation

Now we do what we promised earlier.

- While free registers remain, use the register machine strategy for compilation.

Two-phase strategy translation

Now we do what we promised earlier.

- While free registers remain, use the register machine strategy for compilation.
- When the limit is reached (ie. when there is one register left), revert to the accumulator strategy, using the last register as the accumulator.

Two-phase strategy translation

Now we do what we promised earlier.

- While free registers remain, use the register machine strategy for compilation.
- When the limit is reached (ie. when there is one register left), revert to the accumulator strategy, using the last register as the accumulator.

For this to be work, we need a **register machine** with at least one register that can act as the accumulator. On all popular CPUs (x86, ARM, RISC-V, MIPS), any of the general purpose registers (i.e. not SP, PC) can serve as accumulator.

Code generator with limited registers

The translation for expressions has the following signature:

```
def codegenExp( e : Expr, target :  Register )
          : List [ Instructions ] = ...
```

Code generator with limited registers

The translation for expressions has the following signature:

```
def codegenExp( e : Expr, target :  Register )
          : List [ Instructions ] = ...
```

The result of evaluating `e` should be left in register `target`, and all registers **below** `target` must be unchanged.

Code generator with limited registers

```
def codegenExp( e : Expr, target : Register ) =
  if e is of form
     Ident( x ) then List ( I_Load ( target, x ) )
     Const( n ) then List ( I_LoadImm ( target, n ) )
     Binop( lhs, op, rhs ) then
      if ( target < maxRegs-1 ) // > 1 registers left
        codegenExp ( rhs, target ) ++
        codegenExp ( lhs, target+1 ) ++
        codegenBinop ( op, target, target+1 )
      else // 1 register left, use  as accumulator
        codegenExp ( rhs, target ) ++
        List ( I_Push ( target ) ) ++
        codegenExp ( lhs, target ) ++
        codegenBinopStack ( op, target )
    ...
```

Here **maxRegs** is the number of registers.

Code generator with limited registers

```
def codegenExp( e : Expr, target : Register ) =
  if e is of form
     Ident( x ) then List ( I_Load ( target, x ) )
     Const( n ) then List ( I_LoadImm ( target, n ) )
     Binop( lhs, op, rhs ) then
      if ( target < maxRegs-1 ) // > 1 registers left
        codegenExp ( rhs, target ) ++
        codegenExp ( lhs, target+1 ) ++
        codegenBinop ( op, target, target+1 )
      else // 1 register left, use  as accumulator
        codegenExp ( rhs, target ) ++
        List ( I_Push ( target ) ) ++
        codegenExp ( lhs, target ) ++
        codegenBinopStack ( op, target )
    ...
```

Here **maxRegs** is the number of registers.

Question: Which register is used as accumulator?

Code generator with limited registers

```
def codegenExp( e : Expr, target : Register ) =
  if e is of form
     Ident( x ) then List ( I_Load ( target, x ) )
     Const( n ) then List ( I_LoadImm ( target, n ) )
     Binop( lhs, op, rhs ) then
      if ( target < maxRegs-1 ) // > 1 registers left
        codegenExp ( rhs, target ) ++
        codegenExp ( lhs, target+1 ) ++
        codegenBinop ( op, target, target+1 )
      else // 1 register left, use  as accumulator
        codegenExp ( rhs, target ) ++
        List ( I_Push ( target ) ) ++
        codegenExp ( lhs, target ) ++
        codegenBinopStack ( op, target )
    ...
```

Here **maxRegs** is the number of registers.
Question: Which register is used as accumulator?
Answer: register **maxRegs-1**

Code generator with limited register

```
def codegenBinop( op : Op, r1 : Register, r2 : Register
  if match is of form
     Plus then List ( I_Plus ( r1, r2 ) )
     Minus then List ( I_Minus ( r1, r2 ) )
      ...

def codegenBinopStack( op : Op, r : Register ) =
  if match is of form
     Plus then List ( I_PlusStack ( r ) )
     Minus then List ( I_MinusStack ( r ) )
     ...
```

Note that this 'mixes' the machine code of the register machine (e.g. **`I_Plus`**) with the machine code of the accumulator machine (e.g. **`I_PlusStack`**).

Semantics of the new commands (example)

For example the new command

```
PlusStack r
```

simply does

```
r := r + mem ( SP );
SP := ( SP + 4 ); // each register is 4-byte long
```

For simplicity we are ignoring under/overflowing the stack.

Summary

Summary

We saw a simple code generator for a simple language of statements and expressions, targeting a simple stack machine. This week we looked at ways of improving code for expressions. This involved register machines and accumulator machines, and other addressing modes.

Summary

We saw a simple code generator for a simple language of statements and expressions, targeting a simple stack machine. This week we looked at ways of improving code for expressions. This involved register machines and accumulator machines, and other addressing modes.

Using just one register (accumulator) reduces the amount of memory accesses.

Summary

We saw a simple code generator for a simple language of statements and expressions, targeting a simple stack machine. This week we looked at ways of improving code for expressions. This involved register machines and accumulator machines, and other addressing modes.

Using just one register (accumulator) reduces the amount of memory accesses.

With a large number of registers, the situation gets even better.

Summary

We saw a simple code generator for a simple language of statements and expressions, targeting a simple stack machine. This week we looked at ways of improving code for expressions. This involved register machines and accumulator machines, and other addressing modes.

Using just one register (accumulator) reduces the amount of memory accesses.

With a large number of registers, the situation gets even better.

If you run out out of registers, we can revert to the accumulator scheme.

Summary

We saw a simple code generator for a simple language of statements and expressions, targeting a simple stack machine. This week we looked at ways of improving code for expressions. This involved register machines and accumulator machines, and other addressing modes.

Using just one register (accumulator) reduces the amount of memory accesses.

With a large number of registers, the situation gets even better.

If you run out out of registers, we can revert to the accumulator scheme.

For more sophisticated source languages (if/then/else, while, procedures) this is more tricky. Other approaches to registers are needed (e.g. graph colouring).