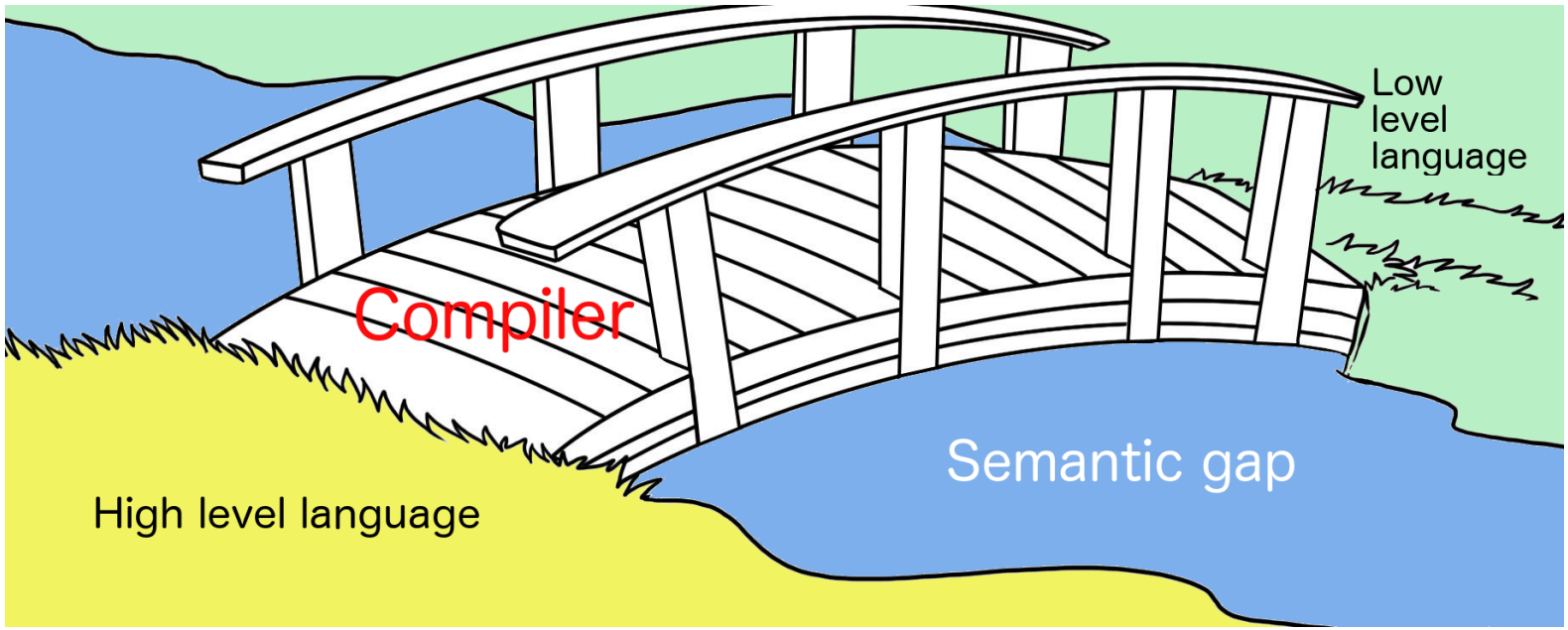
Compilers and computer architecture
Code-generation (3): accumulator-machines

Martin Berger [1]

November 2019

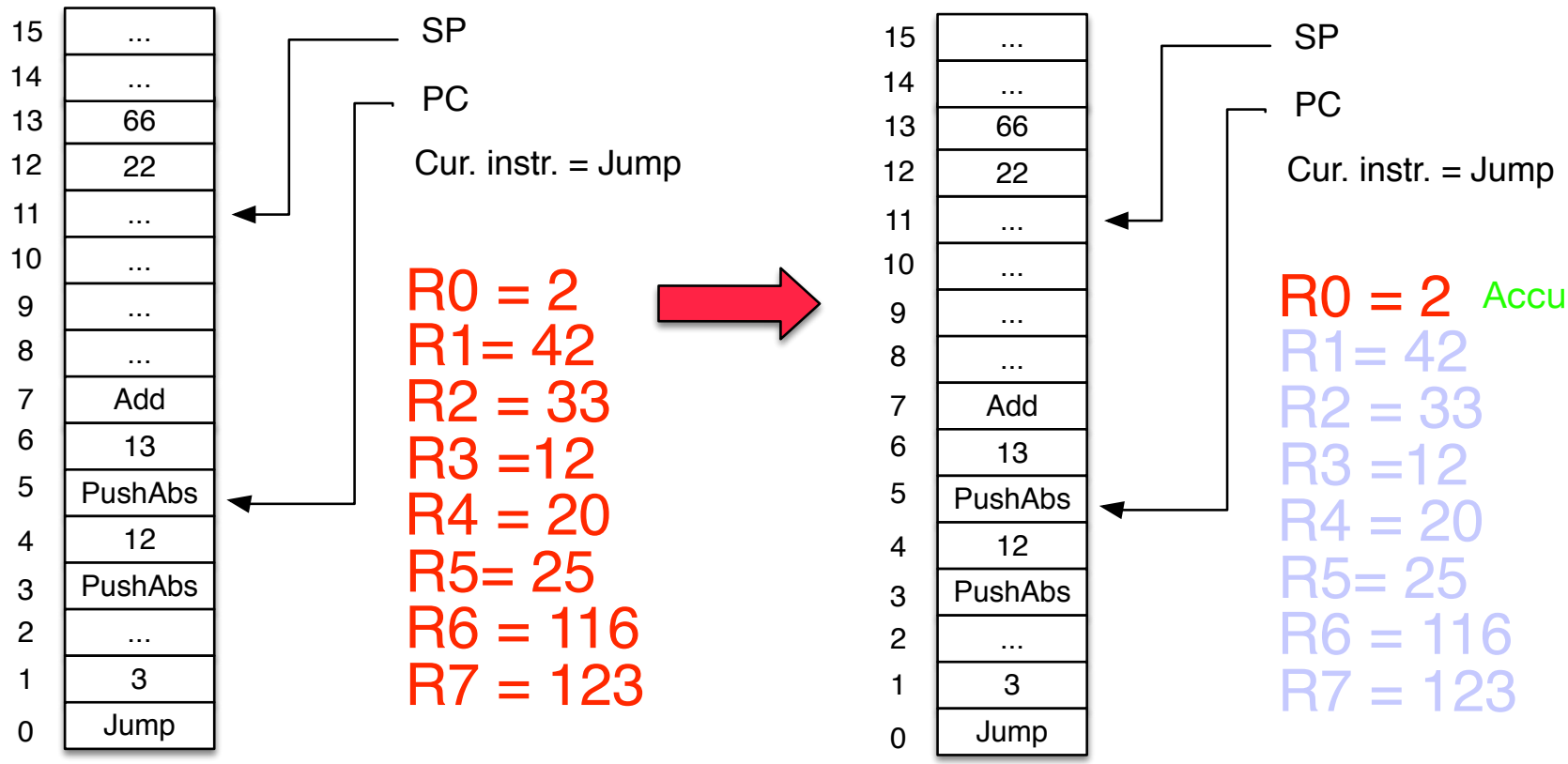
[1] Email: M.F.Berger@sussex.ac.uk, Office hours: Wed 12-13 in Chi-2R312

Recall the function of compilers



The accumulator machine

The accumulator machine



The accumulator machine

The accumulator machine

This machine has a stack, and just one register, the accumulator.

The accumulator machine

This machine has a stack, and just one register, the accumulator.

- For unary operations works like a register machine, e.g.

```
Acc := negate Acc
```

The accumulator machine

This machine has a stack, and just one register, the accumulator.

- For unary operations works like a register machine, e.g.

```
Acc := negate Acc
```

- For binary operations, first argument in accumulator, second argument on the stack, e.g.

```
Acc := Acc + Store[SP]
SP := SP+1
```

The accumulator machine

This machine has a stack, and just one register, the accumulator.

- For unary operations works like a register machine, e.g.

```
Acc := negate Acc
```

- For binary operations, first argument in accumulator, second argument on the stack, e.g.

```
Acc := Acc + Store[SP]
SP := SP+1
```

For simplicity we are ignoring underflowing the stack.

Commands for accumulator machines

Commands for accumulator machines

As with the stack machine, we have PC, SP, IR. In addition we have the “accumulator” (accu) which is a register (i.e. fast).

Commands for accumulator machines

As with the stack machine, we have PC, SP, IR. In addition we have the “accumulator” (accu) which is a register (i.e. fast).

Nop	Does nothing
Pop	removes the top of the stack and stores it in the accumulator
Push	Pushes the content of the accumulator on stack
Load x	Loads the content of memory location x into accumulator
LoadImm n	Loads integer n into accumulator
Store x	Stores the content of accumulator in memory location x
CompGreaterThan	Compares the accumulator with the top of the stack, stores 1 in accumulator if former is bigger than latter otherwise stores 0 there, cleans up stack (i.e. removes top element)

Commands for accumulator machines

CompEq	Like CompGreaterThan but compares for equality
Jump l	Jumps to l
JumpTrue l	Jumps to address/label l if accumulator is not 0
PlusStack	Adds the content of the accumulator with the top of stack storing result in accu, cleans up stack

...

Semantics of accumulator machine

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Push does

```
SP := SP-1; mem ( SP ) := accu
```

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Push does

```
SP := SP-1; mem ( SP ) := accu
```

Pop does

```
accu := mem ( SP ); SP := SP+1
```

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Push does

```
SP := SP-1; mem ( SP ) := accu
```

Pop does

```
accu := mem ( SP ); SP := SP+1
```

Load x does

```
accu := mem ( x )
```

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Push does

```
SP := SP-1; mem ( SP ) := accu
```

Pop does

```
accu := mem ( SP ); SP := SP+1
```

Load x does

```
accu := mem ( x )
```

Store x does

```
mem ( x ) := accu
```

Semantics of accumulator machine

Add does

```
accu := accu + mem (SP); SP:=SP+1
```

Push does

```
SP := SP-1; mem ( SP ) := accu
```

Pop does

```
accu := mem ( SP ); SP := SP+1
```

Load x does

```
accu := mem ( x )
```

Store x does

```
mem ( x ) := accu
```

Etc.

Source language

Source language

Our source language is unchanged: a really simple imperative language.

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

Everything that's difficult to compile, e.g. procedures, objects, is left out. We come to that soon.

Code generation for the accumulator machine

Code generation for the accumulator machine

Code generator for statements looks familiar. Overall structure:

Code generation for the accumulator machine

Code generator for statements looks familiar. Overall structure:

```
def codegen ( s : AST ) =
  if s is of form
    Sequence ( lhs, rhs ) then ...
    Assign ( x, rhs ) then ...
    For ( loopVar, from, to, body ) then ...
```

Translation of Sequencing

Translation of Sequencing

```
def codegen ( s : AST ) =
  if s is of form
    Sequence ( lhs, rhs ) then
       codegen ( lhs ) ++ codegen ( rhs )
       ...
```

Note that ++ is list concatenation

Translation of Assignment $\mathtt{x} \;\; \mathtt{:=} \;\; \mathtt{E}$

Translation of Assignment `x := E`

The code generation for assignment is similar to its translation to the stack machine:

Translation of Assignment `x := E`

The code generation for assignment is similar to its translation to the stack machine:

```
def codegen ( s : AST ) =
  if s is of form
    Assign ( x, rhs ) then
       codegenExpr ( rhs ) ++
       List ( I_Store, I_ConstInt ( x ) )
    ...
```

As before we assume (for now) that we have a code generator for expressions.

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

Translation of Assignment `x := E`

The code generation for assignment is similar to its translation to the stack machine:

```
def codegen ( s : AST ) =
  if s is of form
    Assign ( x, rhs ) then
       codegenExpr ( rhs ) ++
       List ( I_Store, I_ConstInt ( x ) )
    ...
```

As before we assume (for now) that we have a code generator for expressions.

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

What is the semantics of the code for expressions?

Translation of Assignment `x := E`

The code generation for assignment is similar to its translation to the stack machine:

```
def codegen ( s : AST ) =
  if s is of form
    Assign ( x, rhs ) then
       codegenExpr ( rhs ) ++
       List ( I_Store, I_ConstInt ( x ) )
    ...
```

As before we assume (for now) that we have a code generator for expressions.

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

What is the semantics of the code for expressions? The result of computing the translated expression at run-time is left **in the accumulator**.

Translation of expressions

Translation of expressions

```
def codegenExpr ( exp : Expr ) =
    if exp is of form
       Binop ( lhs, op, rhs ) then {
          codegenExpr ( rhs ) ++
          List ( I_Push ) ++
          codegenExpr ( lhs ) ++
          codegenBinop ( op ) }
       Ident ( x ) then List ( I_Load ( x ) )
       Const ( n ) then List ( I_LoadImm ( n ) )
```

Translation of expressions

```
def codegenExpr ( exp : Expr ) =
    if exp is of form
       Binop ( lhs, op, rhs ) then {
          codegenExpr ( rhs ) ++
          List ( I_Push ) ++
          codegenExpr ( lhs ) ++
          codegenBinop ( op ) }
       Ident ( x ) then List ( I_Load ( x ) )
       Const ( n ) then List ( I_LoadImm ( n ) )
```

Here translations of binary operations is as follows.

```
def codegenBinop ( op : Op ) =
    if op is of form
       Plus then List ( I_PlusStack  )
       Minus  then List ( I_MinusStack  )
       ...
```

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

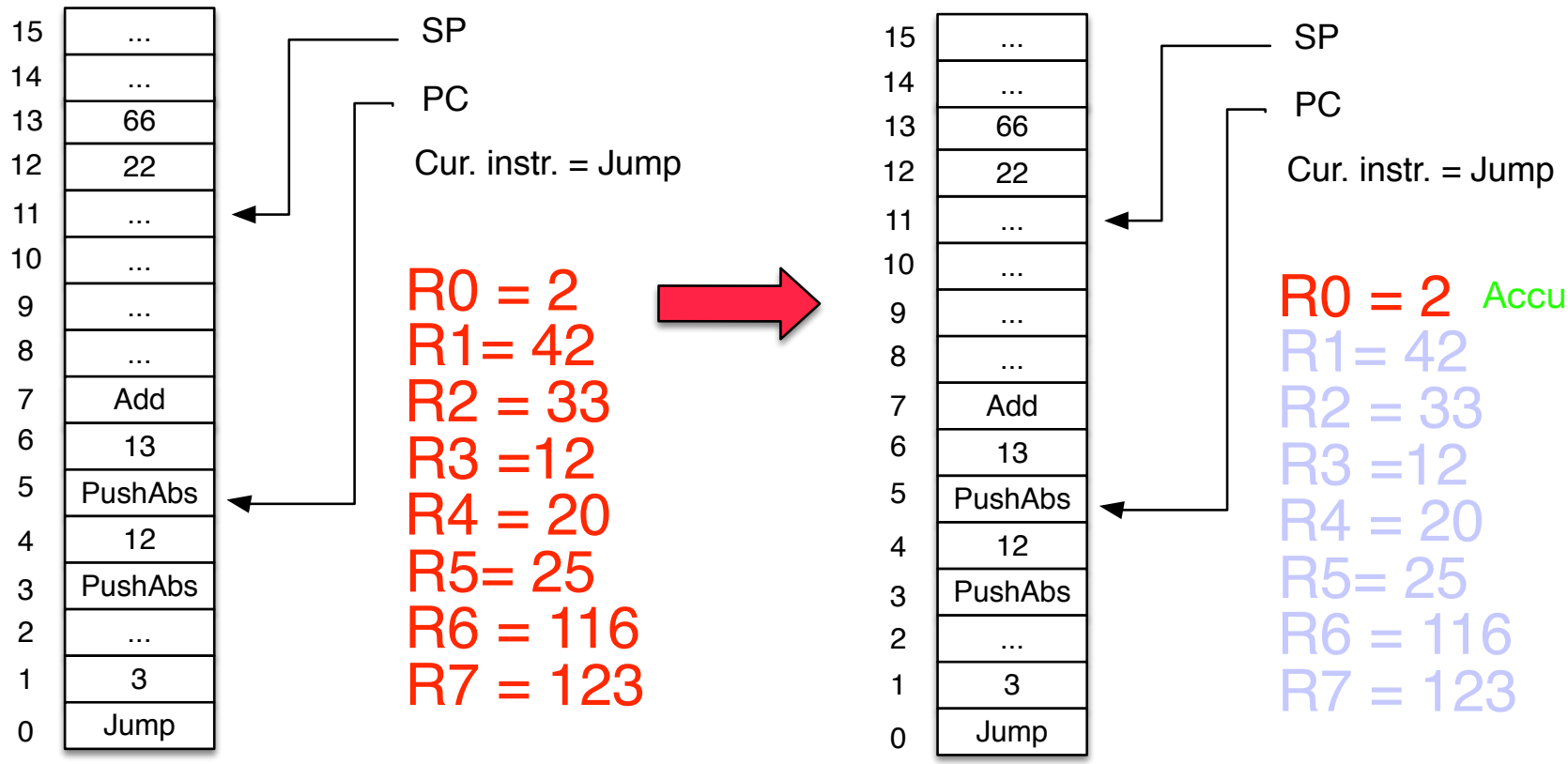
Now we do what we promised earlier.

- 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 workable, we need a **register machine** with 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.

Two-phase strategy translation

Two-phase strategy translation



Code generator with limited registers

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

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.

Check that this is in fact the case.

Code generator with limited 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.

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:

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

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 'merges' the machine language of the register machine (e.g. `I_Plus`) with the language of the accumulator machine (e.g. `I_PlusStack`). In the tutorials, you will be asked to make this precise.

Semantics of the new commands (example)

Semantics of the new commands (example)

For example the new command

```
PlusStack r
```

simply does

```
r := r + mem ( SP );
SP := ( SP + 1 )
```

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).

The material in the textbooks

The material in the textbooks

Sorry, but the material I've presented is not discussed in the form I have used here in the textbooks. Of course all textbooks discuss code generation.