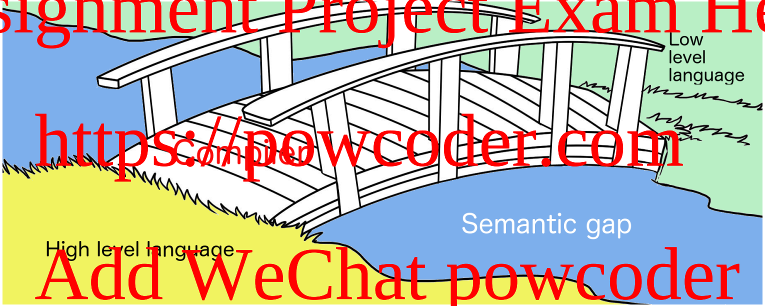
Compilers and computer architecture: A realistic compiler to MIPS

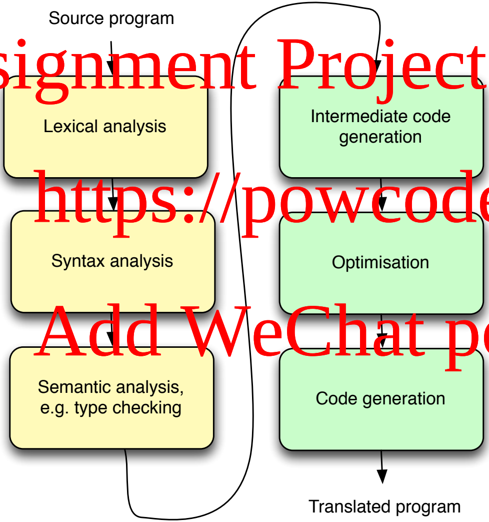
Martin Berger

November 2017

Recall the function of compilers



Recall the structure of compilers



Introduction

Now we look at more realistic code generation. In the previous two lectures we investigated key issues in compilation for more realistic source and target languages, such as procedures and memory alignment. We also introduced the MIPS architecture, which has an especially clean instruction set architecture, is widely used (embedded systems, PS2, PSP), and has deeply influenced other CPU architectures (e.g. ARM).

Source language

The language we translate to MIPS is a simple imperative language with integers as sole data type and **recursive** procedures with arguments. Here's its grammar.

$$
\begin{array}{lcl}
P & \rightarrow & D\, P \mid D \\
D & \rightarrow & \texttt{def}\ \mathit{ID}\,(A) = E \\
A & \rightarrow & A_{ne} \mid \epsilon \\
A_{ne} & \rightarrow & \mathit{ID}, A_{ne} \mid \mathit{ID} \\
E & \rightarrow & \mathit{INT} \mid \mathit{ID} \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & E + E \mid E - E \mid \mathit{ID}(EA) \mid x := E \\
EA & \rightarrow & \epsilon \mid EA_{ne} \\
EA_{ne} & \rightarrow & E \mid E, EA_{ne}
\end{array}
$$

Here ID ranges over identifiers, and INT over integers.

Source language

The language we translate to MIPS is a simple imperative language with integers as sole data type and **recursive** procedures with arguments. Here's its grammar.

$$
\begin{array}{lcl}
P & \rightarrow & D\,P \mid D \\
D & \rightarrow & \texttt{def}\ ID\,(A) = E \\
A & \rightarrow & A_{ne} \mid \epsilon \\
A_{ne} & \rightarrow & ID, A_{ne} \mid ID \\
E & \rightarrow & INT \mid ID \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\
 & \mid & E + E \mid E - E \mid ID(EA) \mid x := E \\
EA & \rightarrow & \epsilon \mid EA_{ne} \\
EA_{ne} & \rightarrow & E \mid E, EA_{ne}
\end{array}
$$

Here *ID* ranges over identifiers, and *INT* over integers. The **first** declared procedure is the entry point (i.e. will be executed when the program is run) and must take **0 arguments**. Procedure names must be **distinct**.

Source language

The language we translate to MIPS is a simple imperative language with integers as sole data type and **recursive** procedures with arguments. Here's its grammar.

$$
\begin{array}{lcl}
P & \rightarrow & D\,P \mid D \\
D & \rightarrow & \texttt{def}\ ID\,(A) = E \\
A & \rightarrow & A_{ne} \mid \epsilon \\
A_{ne} & \rightarrow & ID, A_{ne} \mid ID \\
E & \rightarrow & INT \mid ID \mid \texttt{if}\ E = E\ \texttt{then}\ E\ \texttt{else}\ E \\
& \mid & E + E \mid E - E \mid ID(EA) \mid x := E \\
EA & \rightarrow & \epsilon \mid EA_{ne} \\
EA_{ne} & \rightarrow & E \mid E, EA_{ne}
\end{array}
$$

Here *ID* ranges over identifiers, and *INT* over integers. The **first** declared procedure is the entry point (i.e. will be executed when the program is run) and must take **0 arguments.** Procedure names must be **distinct.**

All variables are of type integer and procedures return integers. We assume that the program passed semantic analysis.

Example program

```
def myFirstProg () = fib ( 3 );

def fib ( n ) =
   if n = 0 then
      1
   else if n = 1 then
      1
   else
      fib ( n-1 ) + fib ( n-2 )
```

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Recall that in an accumulator machine all operations:

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Recall that in an accumulator machine all operations:

- the first argument is assumed to be in the accumulator;

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Recall that in an accumulator machine all operations:

- the first argument is assumed to be in the accumulator;
- all remaining arguments sit on the (top of the) stack;

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Recall that in an accumulator machine all operations:

- the first argument is assumed to be in the accumulator;
- all remaining arguments sit on the (top of the) stack;
- the result of the operation is stored in the accumulator;

Generating code for the language

We use MIPS as an accumulator machine. So we are using only a **tiny** fraction of MIPS's power. This is to keep the compiler easy.

Recall that in an accumulator machine all operations:

- the first argument is assumed to be in the accumulator;
- all remaining arguments sit on the (top of the) stack;
- the result of the operation is stored in the accumulator;
- after finishing the operation, all arguments are removed from the stack.

The code generator we will be presenting guarantees that all these assumptions always hold.

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

- We use the general purpose register `$a0` as accumulator.

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

- We use the general purpose register `$a0` as accumulator.
- We use the general purpose register `$sp` as stack pointer.

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

- We use the general purpose register `$a0` as accumulator.
- We use the general purpose register `$sp` as stack pointer.
- The stack pointer always points to the first free byte above the stack.

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

- We use the general purpose register `$a0` as accumulator.
- We use the general purpose register `$sp` as stack pointer.
- The stack pointer always points to the first free byte above the stack.
- The stack grows downwards.

Generating code for the language

To use MIPS as an accumulator machine we need to decide what registers to use as stack pointer and accumulator.

We make the following assumptions (which are in line with the assumptions the MIPS community makes, see previous lecture slides).

- We use the general purpose register `$a0` as accumulator.
- We use the general purpose register `$sp` as stack pointer.
- The stack pointer always points to the first free byte above the stack.
- The stack grows downwards.

We could have made other choices.

Assumption about data types

Our source language has integers.

Assumption about data types

Our source language has integers.

We will translate them to the built-in 32 bit MIPS data-type.

Assumption about data types

Our source language has integers.

We will translate them to the built-in 32 bit MIPS data-type.

Other choices are possible (e.g. 64 bits, infinite precision, 16 bit etc). This one is by far the simplest.

Assumption about data types

Our source language has integers.

We will translate them to the built-in 32 bit MIPS data-type.

Other choices are possible (e.g. 64 bits, infinite precision, 16 bit etc). This one is by far the simplest.

For simplicity, we won't worry about over/underflow of arithmetic operations.

Code generation

Let's start easy and generate code expressions.

Code generation

Let's start easy and generate code expressions.

For simplicity we'll ignore some issues like placing alignment commands.

Code generation

Let's start easy and generate code expressions.

For simplicity we'll ignore some issues like placing alignment commands.

As with the translation to an idealised accumulator machine a few weeks ago, we compile expressions by recursively walking the AST. We want to write the following:

```
def genExp ( e : Exp ) =
  if e is of form
    IntLiteral ( n ) then ...
    Variable ( x ) then ...
    If ( cond , thenBody, elseBody ) then ...
    Add ( l, r ) then ...
    Sub ( l, r ) then ...
    Call ( f, args ) then ... } }
```

Code generation: integer literals

Let's start with the simplest case.

```
def genExp ( e : Exp ) =
  if e is of form
    IntLiteral ( n ) then
      li $a0 n
```

Code generation: integer literals

Let's start with the simplest case.

```
def genExp ( e : Exp ) =
  if e is of form
    IntLiteral ( n ) then
      li $a0 n
```


Convention: code in red is MIPS code to be executed at run-time. Code in black is compiler code. We are also going to be a bit sloppy about the datatype MIPS_I of MIPS instructions.

Code generation: integer literals

Let's start with the simplest case.

```
def genExp ( e : Exp ) =
  if e is of form
    IntLiteral ( n ) then
      li $a0 n
```


Convention: code in red is MIPS code to be executed at run-time. Code in black is compiler code. We are also going to be a bit sloppy about the datatype MIPS_I of MIPS instructions.

This preserves all invariants to do with the stack and the accumulator as required. Recall that li is a pseudo instruction and will be expanded by the assembler into several real MIPS instructions.

Code generation: addition

```
def genExp ( e : Exp ) =
  if e is of form
    Add ( l, r ) then
      genExp ( l )
      sw $a0 0($sp)
      addiu $sp $sp -4
      genExp ( r )
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addiu $sp $sp 4
```


Note that this evaluates from left to right! Recall also that the stack grows downwards and that the stack pointer points to the first free memory cell above the stack.

Code generation: addition

```
def genExp ( e : Exp ) =
  if e is of form
    Add ( l, r ) then
      genExp ( l )
      sw $a0 0($sp)
      addiu $sp $sp -4
      genExp ( r )
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addiu $sp $sp 4
```


Note that this evaluates from left to right! Recall also that the stack grows downwards and that the stack pointer points to the first free memory cell above the stack.

Question: Why not store the result of compiling the left argument directly in `$t0`?

Code generation: addition

```
def genExp ( e : Exp ) =
  if e is of form
    Add ( l, r ) then
      genExp ( l )
      sw $a0 0($sp)
      addiu $sp $sp -4
      genExp ( r )
      lw $t1 4($sp)
      add $a0 $t1 $a0
      addiu $sp $sp 4
```

Note that this evaluates from left to right! Recall also that the stack grows downwards and that the stack pointer points to the first free memory cell above the stack.

Question: Why not store the result of compiling the left argument directly in `$t0`? Consider 1+(2+3)

Code generation: minus

We want to translate $e - e'$. We need new MIPS command:

```
sub reg1 reg2 reg3
```


It subtracts the content of `reg3` from the content of `reg2` and stores the result in `reg1`. I.e. `reg1 := reg2 - reg3`.

Code generation: minus


```
def genExp ( e : Exp ) =
  if e is of form
    Minus ( l, r ) then
      genExp ( l )
      sw $a0 0($sp)
      addiu $sp $sp -4
      genExp ( r )
      lw $t1 4($sp)
      sub $a0 $t1 $a0 // only change from addition
      addiu $sp $sp 4
```


Note that `sub $a0 $t1 $a0` deducts `$a0` from `$t1`.

Code generation: conditional

We want to translate `if` $e_1 = e_2$ `then` e `else` e'. We need two new MIPS commands:


```
beq reg1 reg2 label
```

```
b label
```


`beq` branches (= jumps) to `label` if the content of `reg1` is identical to the content of `reg2`. Otherwise it does nothing and moves on to the next command.

Code generation: conditional

We want to translate `if` $e_1 = e_2$ `then` e `else` e'. We need two new MIPS commands:


```
beq reg1 reg2 label
```



```
b label
```

`beq` branches (= jumps) to `label` if the content of `reg1` is identical to the content of `reg2`. Otherwise it does nothing and moves on to the next command.

In contrast `b` makes an unconditional jump to `label`.

Code generation: conditional

```
def genExp ( e : Exp ) =
   if e is of form
    If ( l, r, thenBody, elseBody ) then
      val elseBranch = newLabel () // not needed
      val thenBranch = newLabel ()
      val exitLabel = newLabel ()
        genExp ( l )
        sw $a0 0($sp)
        addiu $sp $sp -4
        genExp ( r )
        lw $t1 4($sp)
        addiu $sp $sp 4
        beq $a0 $t1 thenBranch
     elseBranch + ":"
        genExp ( elseBody )
        b exitLabel
     thenBranch +  ":"
        genExp ( thenBody )
     exitLabel +  ":" }
```

Code generation: conditional

```
def genExp ( e : Exp ) =
   if e is of form
    If ( l, r, thenBody, elseBody ) then
      val elseBranch = newLabel () // not needed
      val thenBranch = newLabel ()
      val exitLabel = newLabel ()
        genExp ( l )
        sw $a0 0($sp)
        addiu $sp $sp -4
        genExp ( r )
        lw $t1 4($sp)
        addiu $sp $sp 4
        beq $a0 $t1 thenBranch
     elseBranch + ":"
        genExp ( elseBody )
        b exitLabel
     thenBranch +  ":"
        genExp ( thenBody )
     exitLabel +  ":" }
```

`newLabel` returns new, distinct string every time it is called.

Code generation: procedure calls/declarations

The code a compiler emits for procedure calls and declarations depends on the layout of the activation record (AR).

Code generation: procedure calls/declarations

The code a compiler emits for procedure calls and declarations depends on the layout of the activation record (AR).

The AR stores all the data that's needed to execute an invocation of a procedure.

Code generation: procedure calls/declarations

The code a compiler emits for procedure calls and declarations depends on the layout of the activation record (AR).

The AR stores all the data that's needed to execute an invocation of a procedure.

ARs are held on the stack, because procedure entries and exits are adhere to a bracketing discipline.

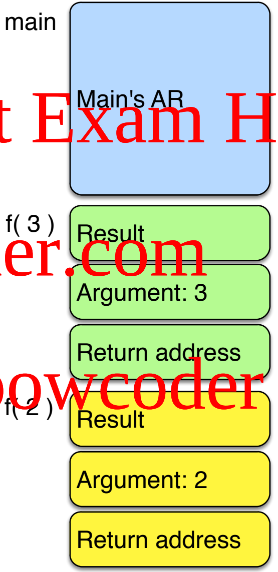
Code generation: procedure calls/declarations

The code a compiler emits for procedure calls and declarations depends on the layout of the activation record (AR).

The AR stores all the data that's needed to execute an invocation of a procedure.

ARs are held on the stack, because procedure entries and exits are adhere to a bracketing discipline.

Note that invocation result and (some) procedure arguments are often passed in register not in AR (for efficiency)



Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

The only variables in the language are procedure parameters. We hold them in AR: for the procedure call $\texttt{f}(e_1, ..., e_n)$ just push the result of evaluating $e_1, ..., e_n$ onto the stack.

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

The only variables in the language are procedure parameters. We hold them in AR: for the procedure call `f`$(e_1, ..., e_n)$ just push the result of evaluating $e_1, ..., e_n$ onto the stack.

The AR needs to store the return address.

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

The only variables in the language are procedure parameters. We hold them in AR: for the procedure call `f`$(e_1, ..., e_n)$ just push the result of evaluating $e_1, ..., e_n$ onto the stack.

The AR needs to store the return address.

The stack calling discipline ensures that on procedure exit `$sp` is the same as on procedure entry.

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

The only variables in the language are procedure parameters. We hold them in AR: for the procedure call $\texttt{f}(e_1, ..., e_n)$ just push the result of evaluating $e_1, ..., e_n$ onto the stack.

The AR needs to store the return address.

The stack calling discipline ensures that on procedure exit `$sp` is the same as on procedure entry.

Also: no registers need to be preserved in accumulator machines. Why?

Code generation: procedure calls/declarations

For our simple language, we can make do with a simple AR layout:

The result is always in the accumulator, so no need for to store the result in the AR.

The only variables in the language are procedure parameters. We hold them in AR: for the procedure call `f`($e_1, ..., e_n$) just push the result of evaluating $e_1, ..., e_n$ onto the stack.

The AR needs to store the return address.

The stack calling discipline ensures that on procedure exit `$sp` is the same as on procedure entry.

Also: no registers need to be preserved in accumulator machines. Why? Because no register is used except for the accumulator and `$t0`, and when a procedure is invoked, all previous evaluations of expressions are already discharged or ’tucked away’ on the stack.

Code generation: procedure calls/declarations

So ARs for a procedure with *n* arguments look like this:

caller's FP
argument n
...
argument 1
return address

A pointer to the **top** of current AR (i.e. where the return address sits) is useful (though not necessary) see later. This pointer is called **frame pointer** and lives in register `$fp`. We need to restore the caller's FP on procedure exit, so we store it in the AR upon procedure entry. The FP makes accessing variables easier (see later).

Code generation: procedure calls/declarations

So ARs for a procedure with *n* arguments look like this:

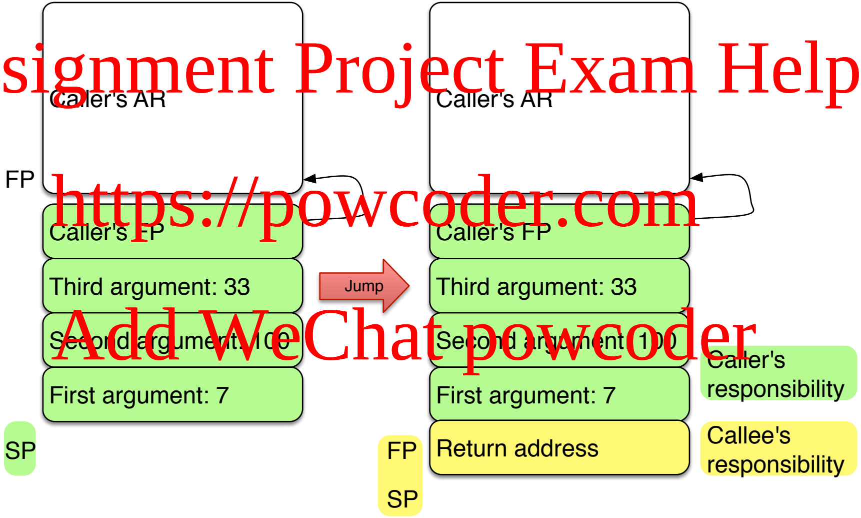
caller's FP
argument n
...
argument 1
return address

A pointer to the **top** of current AR (i.e. where the return address sits) is useful (though not necessary) see later. This pointer is called **frame pointer** and lives in register `$fp`. We need to restore the caller's FP on procedure exit, so we store it in the AR upon procedure entry. The FP makes accessing variables easier (see later).

Arguments are stored in reverse order to make indexing a bit easier.

Code generation: procedure calls/declarations

Let's look at an example: assume we call $f(7, 100, 33)$



Code generation: procedure calls/declarations

To be able to get the return addess for a procedure call easily, we need a new MIPS instruction:


```
jal label
```

Note that `jal` stands for **jump and link.** This instruction does the following:

Code generation: procedure calls/declarations

To be able to get the return addess for a procedure call easily, we need a new MIPS instruction:

```
jal label
```


Note that `jal` stands for **jump and link.** This instruction does the following:

Jumps unconditionally to `label`, stores the address of next instruction (syntactically following `jal label`) in register `$ra`.

Code generation: procedure calls/declarations

To be able to get the return addess for a procedure call easily, we need a new MIPS instruction:


```
jal label
```

Note that `jal` stands for **jump and link.** This instruction does the following:

Jumps unconditionally to `label`, stores the address of next instruction (syntactically following `jal label`) in register `$ra`.

On many other architectures, the return address is automatically placed on the stack by a `call` instruction.

Code generation: procedure calls/declarations

To be able to get the return addess for a procedure call easily, we need a new MIPS instruction:


```
jal label
```

Note that `jal` stands for **jump and link**. This instruction does the following:

Jumps unconditionally to `label`, stores the address of next instruction (syntactically following `jal label`) in register `$ra`.

On many other architectures, the return address is automatically placed on the stack by a `call` instruction.

On MIPS we must push the return address on stack explicitly. This can only be done by callee, because address is available only after `jal` has executed.

Code generation: procedure calls

Example of procedure call with 3 arguments. General case is similar.

Code generation: procedure calls

Example of procedure call with 3 arguments. General case is similar.


```
case Call ( f, List ( e1, e2, e3 ) ) then
  sw $fp 0($sp) // save FP on stack
  addiu $sp $sp -4
  genExp ( e3 ) // we choose right-to-left ev. order
  sw $a0 0($sp) // save 3rd argument on stack
  addiu $sp $sp -4
  genExp ( e2 )
  sw $a0 0($sp) // save 2nd argument on stack
  addiu $sp $sp -4
  genExp ( e1 )
  sw $a0 0($sp) // save 1st argument on stack
  addiu $sp $sp -4
  jal ( f + "_entry" )  // jump to f, save return
                        // addr in $ra
```

Code generation: procedure calls

Several things are worth noting.

Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).

Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).
- Then the caller saves procedure parameters in reverse order (right-to-left).

Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).
- Then the caller saves procedure parameters in reverse order (right-to-left).
- Implicitly the caller saves the return address in `$ra` by executing `jal`. The return address is still not in the AR on the stack. The AR is incomplete. Completion is the callee's responsibility.

Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).
- Then the caller saves procedure parameters in reverse order (right-to-left).
- Implicitly the caller saves the return address in `$ra` by executing `jal`. The return address is still not in the AR on the stack. The AR is incomplete. Completion is the callee's responsibility.
- How big is the AR?

Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).
- Then the caller saves procedure parameters in reverse order (right-to-left).
- Implicitly the caller saves the return address in `$ra` by executing `jal`. The return address is still not in the AR on the stack. The AR is incomplete. Completion is the callee's responsibility.
- How big is the AR? For a procedure with n arguments the AR (without return address) is $4 + 4 * n = 4(n + 2)$ bytes long. This is **know at compile time** and is important for the compilation of procedure bodies.

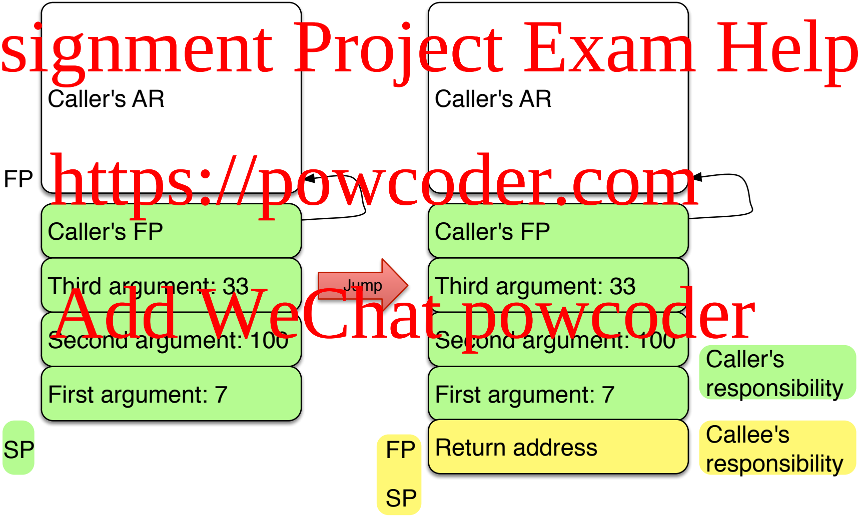
Code generation: procedure calls

Several things are worth noting.

- The caller first saves the FP (i.e. pointer to top of its own AR).
- Then the caller saves procedure parameters in reverse order (right-to-left).
- Implicitly the caller saves the return address in `$ra` by executing `jal`. The return address is still not in the AR on the stack. The AR is incomplete. Completion is the callee's responsibility.
- How big is the AR? For a procedure with n arguments the AR (without return address) is $4 + 4 * n = 4(n + 2)$ bytes long. This is **know at compile time** and is important for the compilation of procedure bodies.
- The translation of procedure invocations is generic in the number of procedure arguments, nothing particular about 3.

Code generation: procedure calls

So far we perfectly adhere to the lhs of this picture (except 33, 100, 7).



Code generation: procedure calls, callee's side

In order to compile a declaration `d` like

```
def f ( x1, ..., xn ) = body
```


we use a procedure for compiling declarations like so:

```
def genDecl ( d ) = ...
```

Code generation: procedure calls, callee's side

We need two new MIPS instructions:


```
jr reg
```

```
move reg reg'
```

Code generation: procedure calls, callee's side

We need two new MIPS instructions:

```
jr reg
```



```
move reg reg'
```

The former (`jr reg`) jumps to the address stored in register `reg`.

Code generation: procedure calls, callee's side

We need two new MIPS instructions:


```
jr reg
```

```
move reg reg'
```

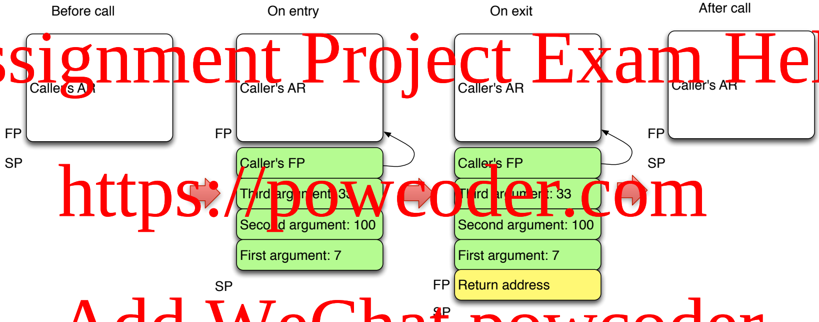

The former (`jr reg`) jumps to the address stored in register `reg`.

The latter (`move reg reg'`) moves the content of register `reg'` into the register `reg`.

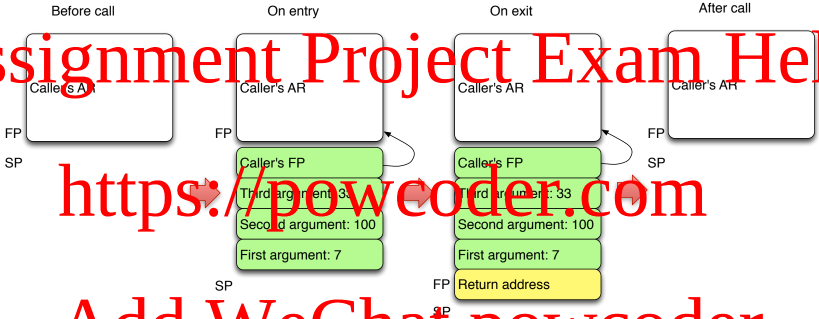
Code generation: procedure calls, callee's side

```
def genDecl ( d : Declaration ) =
  val sizeAR = ( 2 + d.args.size ) * 4
         // each procedure argument takes 4 bytes
         // in addition the AR stores the return
         // address and old FP
  d.id + "_entry:" // label to jump to
    move $fp $sp // FP points to top of current AR
    sw $ra 0($sp) // put return address on stack
    addiu $sp $sp -4  // now AR is fully created
    genExp ( d.body )
    lw $ra 4($sp) // load return address into $ra
                  // could also use $fp
    addiu $sp $sp sizeAR // pop AR off stack in one go
    lw $fp 0($sp) // restore old FP
    jr $ra  // hand back control to caller
```

Code generation: procedure calls, callee's side



Code generation: procedure calls, callee's side



So we preserve the invariant that the stack looks exactly the same before and after a procedure call!

Code generation: frame pointer

Variables are just the procedure parameters in this language.

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Problem: The stack grows and shrinks when intermediate results are computed (in the accumulator machine approach), so the variables are not on a fixed offset from `$sp`. For example in


```
def f ( x, y, z ) = x + ( ( x * z ) + ( y - y ) )
```

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Problem: The stack grows and shrinks when intermediate results are computed (in the accumulator machine approach), so the variables are not on a fixed offset from `$sp`. For example in


```
def f ( x, y, z ) = x + ( ( x * z ) + ( y - y ) )
```


Solution:

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Problem: The stack grows and shrinks when intermediate results are computed (in the accumulator machine approach), so the variables are not on a fixed offset from `$sp`. For example in

```
def f ( x, y, z ) = x + ( ( x * z ) + ( y - y ) )
```

Solution: Use **frame pointer** `$fp`.

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Problem: The stack grows and shrinks when intermediate results are computed (in the accumulator machine approach), so the variables are not on a fixed offset from `$sp`. For example in


```
def f ( x, y, z ) = x + ( ( x * z ) + ( y - y ) )
```


Solution: Use **frame pointer** `$fp`.

- Always points to the top of current AR as long as invocation is active.

Code generation: frame pointer

Variables are just the procedure parameters in this language.

They are all on the stack in the AR, pushed by the caller. How do we access them? The obvious solution (use the SP with appropriate offset) does not work (at least not easily).

Problem: The stack grows and shrinks when intermediate results are computed (in the accumulator machine approach), so the variables are not on a fixed offset from `$sp`. For example in

```
def f ( x, y, z ) = x + ( ( x * z ) + ( y - y ) )
```

Solution: Use **frame pointer** `$fp`.

- Always points to the top of current AR as long as invocation is active.
- The FP does not (appear to) move, so we can find all variables at a fixed offset from `$fp`.

Code generation: variable use

Let's compile *x* which is the i-th (starting to count from 1) parameter of `def f(x1, x2, ..., xn) = body` works like this (using offset in AR):


```
def genExp ( e : Exp ) =
   if e is of form Variable ( x ) then
      val offset = 4*i
      lw $a0 offset($fp)
```

Code generation: variable use

Let's compile *x* which is the i-th (starting to count from 1) parameter of `def f(x1, x2, ..., xn) = body` works like this (using offset in AR):


```
def genExp ( e : Exp ) =
   if e is of form Variable ( x ) then
      val offset = 4*i
      lw $a0 offset($fp)
```


Putting the arguments in reverse order on the stack makes the offseting calculation `val offset = 4*i` a tiny bit easier.

Code generation: variable use

Let's compile *x* which is the i-th (starting to count from 1) parameter of `def f(x1, x2, ..., xn) = body` works like this (using offset in AR):

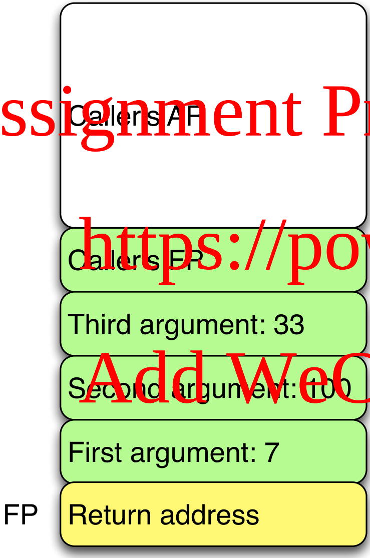

```
def genExp ( e : Exp ) =
   if e is of form Variable ( x ) then
      val offset = 4*i
      lw $a0 offset($fp)
```


Putting the arguments in reverse order on the stack makes the offseting calculation `val offset = 4*i` a tiny bit easier.

Key insight: access at **fixed offset** relative to a dynamically changing pointer. Offset and pointer location are known at compile time.

This idea is pervasive in compilation.

Code generation: variable use



In the declaration `def f (x, y, z) = ...`, we have:

- x is at address `$fp + 4`
- y is at address `$fp + 8`
- z is at address `$fp + 12`

Note that this work because indexing begins at 1 in this case, and arguments are pushed on stack from right to left.

Translation of variable assignment

Given that we know now that reading a variable is translated as

```
if e is of form Variable ( x ) then
   val offset = 4*i
   lw $a0 offset($fp)
```

How would you translate an assignment

```
x := e
```

Translation of variable assignment

Given that we know now that reading a variable is translated as

```
if e is of form Variable ( x ) then
   val offset = 4*i
   lw $a0 offset($fp)
```


How would you translate an assignment

```
x := e
```


Assume that the variable *x* is the i-th (starting to count from 1) formal parameter of the ambient procedure declaration.

Translation of variable assignment

Given that we know now that reading a variable is translated as

```
if e is of form Variable ( x ) then
   val offset = 4*i
   lw $a0 offset($fp)
```

How would you translate an assignment

```
x := e
```

Assume that the variable *x* is the i-th (starting to count from 1) formal parameter of the ambient procedure declaration.

```
def genExp ( exp : Exp ) =
  if exp is of form Assign ( x, e ) then
     val offset = 4*i
     genExp ( e )
     sw $a0 offset($fp)
```

Translation of variable assignment

Given that we know now that reading a variable is translated as

```
if e is of form Variable ( x ) then
   val offset = 4*i
   lw $a0 offset($fp)
```


How would you translate an assignment

```
x := e
```


Assume that the variable *x* is the i-th (starting to count from 1) formal parameter of the ambient procedure declaration.

```
def genExp ( exp : Exp ) =
  if exp is of form Assign ( x, e ) then
     val offset = 4*i
     genExp ( e )
     sw $a0 offset($fp)
```


Easy!

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code and layout also depends on CPU.

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code and layout also depends on CPU.

Code generation happens by recursive AST walk.

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code and layout also depends on CPU.

Code generation happens by recursive AST walk.

Industrial strength compilers are more complicated:

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code and layout also depends on CPU.

Code generation happens by recursive AST walk.

Industrial strength compilers are more complicated:

- Try to keep values in registers, especially the current stack frame. E.g. compilers for MIPS usually pass first four procedure arguments in registers `$a0 - $a3`.

Code generation: summary remarks

The code of variable access, procedure calls and declarations depends totally on the layout of the AR, so the AR must be designed together with the code generator, and all parts of the code generator must agree on AR conventions. It's just as important to be clear about the nature of the stack (grows upwards or downwards), frame pointer etc.

Access at **fixed offset** relative to dynamically changing pointer. Offset and pointer location are known at compile time.

Code and layout also depends on CPU.

Code generation happens by recursive AST walk.

Industrial strength compilers are more complicated:

- Try to keep values in registers, especially the current stack frame. E.g. compilers for MIPS usually pass first four procedure arguments in registers `$a0 - $a3`.
- Intermediate values, local variables are held in registers, not on the stack.

Non-integer procedure arguments

What we have not covered is procedures taking non integer arguments.

Non-integer procedure arguments

What we have not covered is procedures taking non integer arguments.

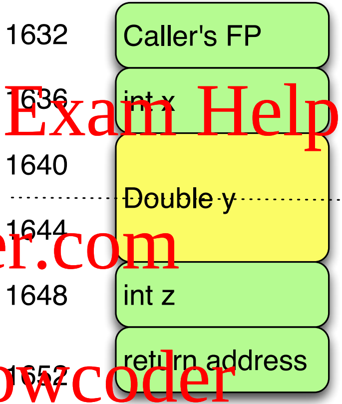
This is easy: the only difference from a code generation perspective between integer types and other types as procedure arguments is the **size** of the data. But that size is known at compile-time (at least for languages that are statically typed). For example the type `double` is often 64 bits. So we reserve 8 bytes for arguments of that type in the procedure's AR layout. We may have to use two calls to `lw` and `sw` to load and store such arguments, but otherwise code generation is unchanged.

Non-integer procedure arguments

Consider a procedure with the following signature:

```
int f ( int x,
        double y,
        int z ) = { ... }
```

(Not valid in our mini-language).
Assuming that `double` is stored as 64 bits, then the AR would look like on the right.

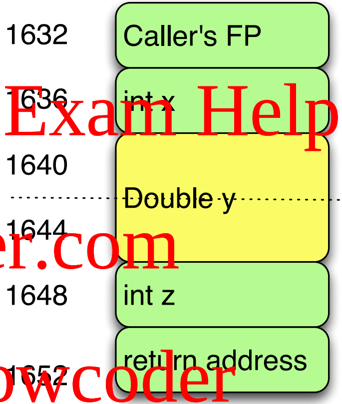


Non-integer procedure arguments

Consider a procedure with the following signature:


```
int f ( int x,
        double y,
        int z ) = { ... }
```


(Not valid in our mini-language).
Assuming that `double` is stored as 64 bits, then the AR would look like on the right.



How does the code generator know what size the variables have?

Non-integer procedure arguments

Consider a procedure with the following signature:

```
int f ( int x,
        double y,
        int z ) = { ... }
```

(Not valid in our mini-language).
Assuming that `double` is stored as 64 bits, then the AR would look like on the right:

1632	Caller's FP
1636	int x
1640	Double y
1644	
1648	int z
1652	return address

How does the code generator know what size the variables have?

Using the information stored in the symbol table, which was created by the type checker and passed to the code-generator.

Non-integer procedure arguments

Due to the simplistic accumulator machine approach, cannot do the same with the return value, e.g.


```
double f ( int x, double y, int z ) = ...
```

Non-integer procedure arguments

Due to the simplistic accumulator machine approach, cannot do the same with the return value, e.g.

```
double f ( int x, double y, int z ) = ...
```

This is because the accumulator holds the return value of procedure calls, and the accumulator is fixed at 32 bits.

Non-integer procedure arguments

Due to the simplistic accumulator machine approach, cannot do the same with the return value, e.g.


```
double f ( int x, double y, int z ) = ...
```


This is because the accumulator holds the return value of procedure calls, and the accumulator is fixed at 32 bits.

In this case we'd have to move to an approach that holds the return value also in the AR (either for all arguments, or only for arguments that don't fit in a register – we know at compile time which is which).

Example `def sumto(n) = if n=0 then 0 else n+sumto(n-1)`


```
sumto_entry:
   move $fp $sp
   sw $ra 0($sp)
   addiu $sp $sp -4
   lw $a0 4($fp)
   sw $a0 0($sp)
   addiu $sp $sp -4
   li $a0 0
   lw $t1 4($sp)
   addiu $sp $sp 4
   beq $a0 $t1 then1
else0:
   lw $a0 4($fp)
   sw $a0 0($sp)
   addiu $sp $sp -4
   sw $fp 0($sp)
   addiu $sp $sp -4
   lw $a0 4($fp)
   sw $a0 0($sp)
   addiu $sp $sp -4
   li $a0 1
   lw $t1 4($sp)
   sub $a0 $t1 $a0
   addiu $sp $sp 4
   sw $a0 0($sp)
   addiu $sp $sp -4
   jal sumto_entry
   lw $t1 4($sp)
   add $a0 $t1 $a0
   addiu $sp $sp 4
   b exit2
then1:
   li $a0 0
exit2:
   lw $ra 4($sp)
   addiu $sp $sp 12
   lw $fp 0($sp)
   jr $ra
```

Several points are worth thinking about.

Interesting observations

Several points are worth thinking about.

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation. We will soon learn about heap-allocation (section on garbage-collection), which is much more expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

Interesting observations

Several points are worth thinking about.

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation. We will soon learn about heap-allocation (section on garbage-collection), which is much more expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

The source language has recursion. The target language (MIPS) does not. What is recursion translated to?

Interesting observations

Several points are worth thinking about.

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation. We will soon learn about heap-allocation (section on garbage-collection), which is much more expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

The source language has recursion. The target language (MIPS) does not. What is recursion translated to? Jumping!

Interesting observations

Several points are worth thinking about.

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation. We will soon learn about heap-allocation (section on garbage-collection), which is much more expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

The source language has recursion. The target language (MIPS) does not. What is recursion translated to? Jumping! But what kind of jumping?

Interesting observations

Several points are worth thinking about.

Stack allocated memory is much **faster** than heap allocation, because (1) acquiring stack memory is just a constant-time push operation, and (2) the whole AR can be 'deleted' (= popped off the stack) in a single, constant-time operation. We will soon learn about heap-allocation (section on garbage-collection), which is much more expensive. This is why low-level language (C, C++, Rust) don't have garbage collection (by default).

The source language has recursion. The target language (MIPS) does not. What is recursion translated to? Jumping! But what kind of jumping? **Backwards jumping**.

Another interesting observation: inefficiency of the translation

As already pointed out at the beginning of this course, stack- and accumulator machines are inefficient.

Another interesting observation: inefficiency of the translation

As already pointed out at the beginning of this course, stack- and accumulator machines are inefficient. Consider this from the previous slide (compilation of parts of `n == 0` in `sumto`):

```
lw $a0 4($fp)       // first we load n into the
                    // accumulator from the stack
sw $a0 0($sp)       // then we push n back onto
                    // the stack
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)       // now we load n back from
                    // the stack into a temporary
```

Another interesting observation: inefficiency of the translation

As already pointed out at the beginning of this course, stack- and accumulator machines are inefficient. Consider this from the previous slide (compilation of parts of `n = 0` in `sumto`):

```
lw $a0 4($fp)        // first we load n into the
                     // accumulator from the stack
sw $a0 0($sp)        // then we push n back onto
                     // the stack
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)        // now we load n back from
                     // the stack into a temporary
```

This is the price we pay for the simplicity of compilation strategy.

Another interesting observation: inefficiency of the translation

As already pointed out at the beginning of this course, stack- and accumulator machines are inefficient. Consider this from the previous slide (compilation of parts of `n = 0` in `sumto`):


```
lw $a0 4($fp)      // first we load n into the
                   // accumulator from the stack
sw $a0 0($sp)      // then we push n back onto
                   // the stack
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)      // now we load n back from
                   // the stack into a temporary
```


This is the price we pay for the simplicity of compilation strategy.

It's possible to do much better, e.g. saving it directly in `$t1` using better compilation strategies and optimisation techniques.

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the 'preamble', e.g. setting up data declarations, alignment commands etc.

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the 'preamble', e.g. setting up data declarations, alignment commands etc.
- Generate code for each declaration.

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the 'preamble', e.g. setting up data declarations, alignment commands etc.
- Generate code for each declaration.
- Emit code enabling the OS to call the first procedure (like Java's `main` – other languages might have different conventions) 'to get the ball rolling'. This essentially involves

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the ’preamble’, e.g. setting up data declarations, alignment commands etc.
- Generate code for each declaration.
- Emit code enabling the OS to call the first procedure (like Java’s `main` – other languages might have different conventions) ’to get the ball rolling’. This essentially involves
  1. Creating (the caller’s side of) an activation record.

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the 'preamble', e.g. setting up data declarations, alignment commands etc.
- Generate code for each declaration.

- Emit code enabling the OS to call the first procedure (like Java's `main` – other languages might have different conventions) 'to get the ball rolling'. This essentially involves:

  1. Creating (the caller's side of) an activation record.
  2. Jump-and-link'ing to the first procedure.

Compiling whole programs

So far we have only compiled expressions and single declarations, but a program is a sequence of declarations, and it is called from, and returns to the OS. To compile a whole program we do the following:

- Creating the 'preamble', e.g. setting up data declarations, alignment commands etc.
- Generate code for each declaration.

- Emit code enabling the OS to call the first procedure (like Java's `main` – other languages might have different conventions) 'to get the ball rolling'. This essentially involves:

  1. Creating (the caller's side of) an activation record.
  2. Jump-and-link'ing to the first procedure.
  3. Code that hands back control gracefully to the OS after program termination. Termination means doing a return to the place after (2). This part is highly OS specific.

Compiling whole programs

Say we had a program declaring 4 procedures `f1`, `f2`, `f3`, and `f4` in this order. Then a fully formed compiler would typically generate code as follows.


```
preamble
   ...// e.g. alignment commands if needed
entry_point: // this is where the OS jumps to
             // at startup
   ... // create AR for initial call to f1
   jal f1_entry // jump to f1
   ... // cleanup, hand back control to OS
f1_entry:
   ... // f1 body code
f2_entry:
   ... // f2 body code
f3_entry:
   ... // f3 body code
f4_entry:
   ... // f4 body code
```