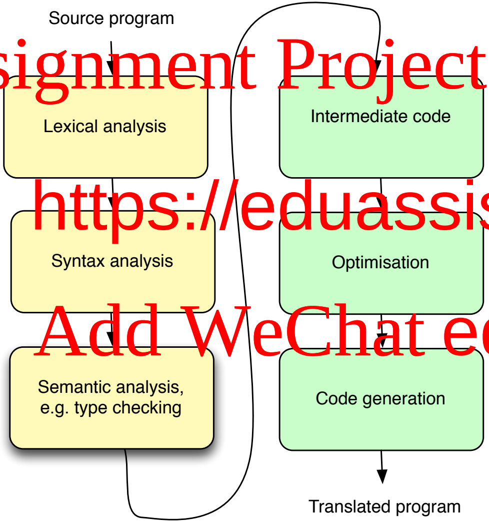
Compilers and computer architecture:
A realistic compiler to MIPS

November 20

Recall the function of compilers

Recall the structure of compilers



Now we look at more realistic code generation. In the previous
two le
reali

me
whic
widely used (embedded systems, PS2, PS
influenced other CPU architectures (e.g. A

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.

$$P \rightarrow D\,P \mid D$$
$$D \rightarrow \texttt{def}\ ID\,(A) = E$$

$$EA \rightarrow \epsilon \mid EA_{ne}$$
$$EA_{ne} \rightarrow E \mid E\ EA_{ne}$$

Here *ID* ranges over identifiers, and

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.

$$P \rightarrow D\,P \mid D$$

$$D \rightarrow \texttt{def}\ ID\,(A) = E$$

$$EA \rightarrow \epsilon \mid EA_{ne}$$

$$EA_{ne} \rightarrow E \mid E, EA_{ne}$$

Here *ID* ranges over identifiers, and

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.

$$
\begin{array}{lcl}
P & \rightarrow & D\; P \mid D \\
D & \rightarrow & \texttt{def}\; ID\,(A) = E
\end{array}
$$

$$
\begin{array}{lcl}
EA & \rightarrow & \epsilon \mid EA_{ne} \\
EA_{ne} & \rightarrow & E \mid E\,,\, EA_{ne}
\end{array}
$$

Here ID ranges over identifiers, and

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 );

        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.

Rec

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.

Rec

-

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.

Rec

-
-

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.

Rec

-
-
- the result of the operation is stored in the a

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.

Rec

-
-

- the result of the operation is stored in the a
- after finishing the operation, all argum 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 slide

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 slide

- ator

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 slide

- ator.
- `$sp` inter.

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 slide

- ator.
- $sp inter.
- The stack pointer always points to the fir the stack.

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 slide

- ator.
- `$sp` inter.
- The stack pointer always points to the fir 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 slide

- ator.
- `$sp` inter.
- The stack pointer always points to the fir 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 w

Assumption about data types

Our source language has integers.

We w

Oth
etc).

Assumption about data types

Our source language has integers.

We w

Oth

etc).

For simplicity, we won't worry about over/un
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 w

the A


```
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
```


Con
run-t
to be a bit sloppy about the datatype
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
```


Con
run-t
to be a bit sloppy about the datatype
instructions.

This preserves all invariants to do with the sta 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

      addiu $sp $sp 4
```


Note that this evaluates from left to right! Reca stack grows downwards and that the stack p 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

      addiu $sp $sp 4
```


Note that this evaluates from left to right! Reca
stack grows downwards and that the stack p
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

      addiu $sp $sp 4
```


Note that this evaluates from left to right! Reca
stack grows downwards and that the stack p
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:

It sub 2 and stores the result in reg1. I.e. reg1 := r

Code generation: minus


```
def genExp ( e : Exp ) =
  if e is of form
    Minus ( l, r ) then
```



```
      lw $t1 4($sp)
      sub $a0 $t1 $a0 // only change from a
      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
```


`beq` branches (= jumps) to `label` if t
identical to the content of `reg2`. Othe
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
```


`beq` branches (= jumps) to `label` if t
identical to the content of `reg2`. Othe
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 ()

        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 ()

      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 sto
to execut

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 sto

to execut

ARs are he

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 sto
to execut

ARs are he
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 o

We h st

pus

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 o

We h st

pus

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 o

We h st

pus

The AR needs to store the return address.

The stack calling discipline ensures that on p 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 o

We h st

pus

The AR needs to store the return address.

The stack calling discipline ensures that on p 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 o

We h st

pus

The AR needs to store the return address.

The stack calling discipline ensures that on p `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
...
arg
ret

A poi
address sits) is useful (though not necessar
pointer is called **frame pointer** and liv
need to restore the caller's FP on procedure e
in the AR upon procedure entry. The FP make
variables easier (see later).

Code generation: procedure calls/declarations

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

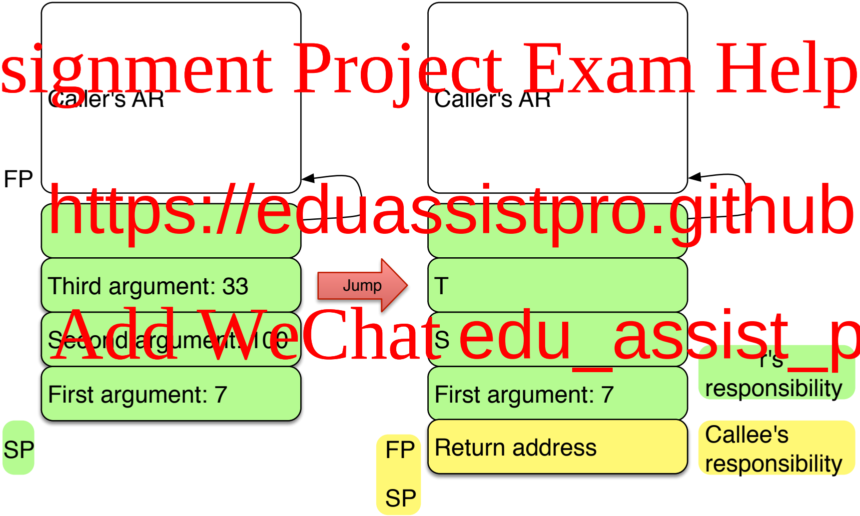
caller's FP
argument n
...
arg
ret

A poi
address sits) is useful (though not necessar
pointer is called **frame pointer** and liv
need to restore the caller's FP on procedure e
in the AR upon procedure entry. The FP make
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 es

the fo

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 es
the fo

Jum
instruction (syntactically following `ja` `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 es
the fo

Jum
instruction (syntactically following ja ra.

On many other architectures, the return add
automatically placed on the stack by a

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 es
the fo

Jum
instruction (syntactically following `ja` `ra`.

On many other architectures the return add
automatically placed on the stack by a

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
  a
  g
  s
  a
  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).
- by 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).
- by

  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).
- by

  responsibility.
- How big is the AR? For a procedu e AR (without return address) is 4 s long. This is **know at compile time** the compilation of procedure bodies.

Code generation: procedure calls

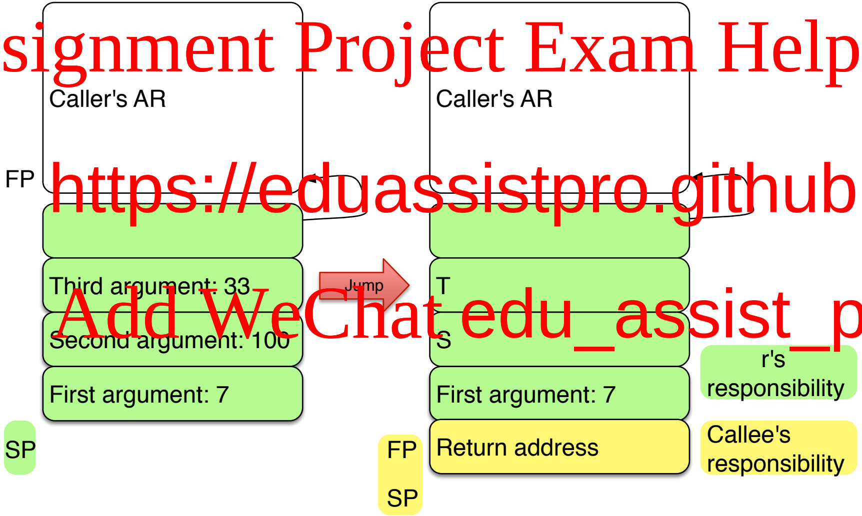
Several things are worth noting.

- 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).
- by

  responsibility.
- How big is the AR? For a procedu e AR (without return address) is 4 s long. This is **know at compile time** 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

```
d
```

we us


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

Code generation: procedure calls, callee's side

We need two new MIPS instructions:


```
jr reg
```

Code generation: procedure calls, callee's side

We need two new MIPS instructions:

```
jr reg
```


The former (`jr reg`) jumps to the addre

`reg`.

Code generation: procedure calls, callee's side

We need two new MIPS instructions:


```
jr reg
```


The former (`jr reg`) jumps to the addre

`reg`.

The latter (`move reg reg'`) moves th

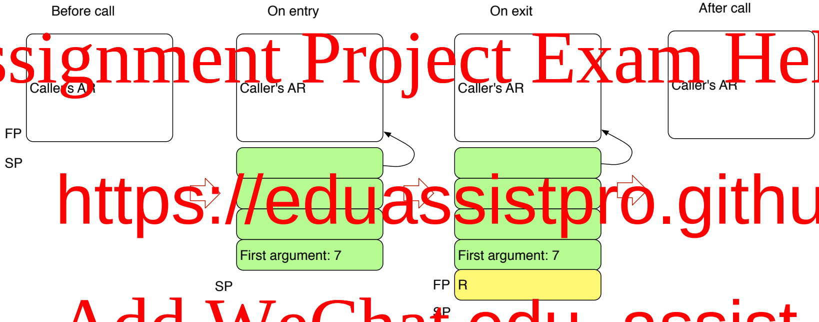
`reg'` into the register `reg`.

```
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
```

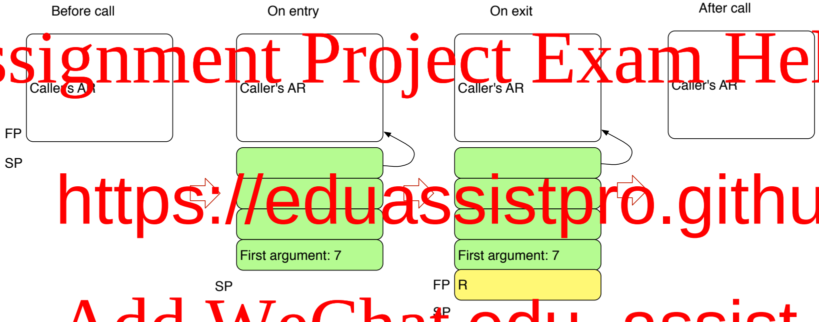


```
    addiu $sp $sp -4  // now AR is fully created
    genExp ( d.body )
    lw $ra 4($sp)  // load return address in
                   // could also use $
    addiu $sp $sp sizeAR // pop AR off stack in o
    lw $fp 0($sp) // restore old FP
    jr $ra  // hand back control to caller
```

Code generation: procedure calls, callee's side



Code generation: procedure calls, callee's side



So we preserve the invariant that the stack loo
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).

Prob
resu
so th mpe
in


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

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).

Prob

resu

so th mpe

in


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


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).

Prob
resu
so th mpe
in


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


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).

Prob
resu
so th
in

mpe

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


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).

Prob
resu
so th mpe
in


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


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
```

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
```


Putting the arguments in reverse order on th

offseting calculation `val offset = 4`

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):

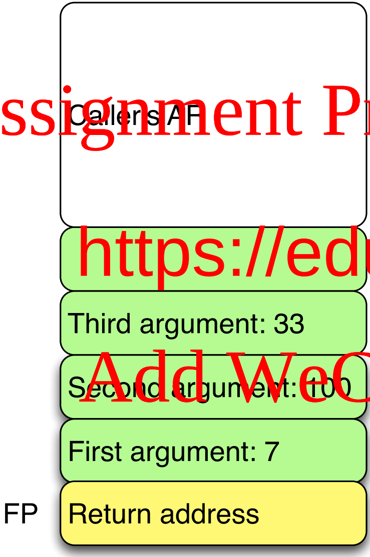

```
def genExp ( e : Exp ) =
   if e is of form Variable ( x ) then
```


Putting the arguments in reverse order on th offseting calculation `val offset = 4`

Key insight: access at **fixed offset** r 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:

Note th
indexi
and ar
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

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

Ass

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

Ass

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

Ass

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.

Acc er.
Offs

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.

Acc [illegible] er.
Offs [illegible]

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.

Acc ... er.
Offs

Code and layout also depends on CPU.

Code generation happens by recursive AS

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.

Acc [illegible] er.
Offs [illegible]

Code and layout also depends on CPU.

Code generation happens by recursive AS [illegible]

Industrial strength compilers are more com [illegible]

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.

Acc … er.
Offs

Code and layout also depends on CPU.

Code generation happens by recursive AS

Industrial strength compilers are more com

- 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.

Acc er.
Offs

Code and layout also depends on CPU.

Code generation happens by recursive AS

Industrial strength compilers are more com

- 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 i
pers
proc
kno
typed). For example the type `doubl`
reserve 8 bytes for arguments of that type in th
AR layout. We may have to use two calls to
and store such arguments, but otherwise co
unchanged.

Non-integer procedure arguments

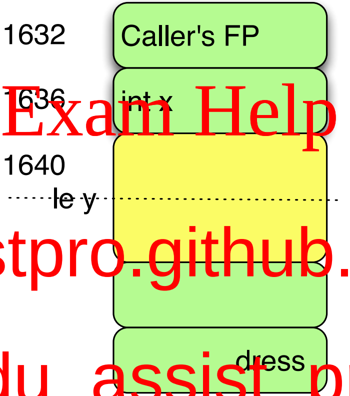
Consider a procedure with the following signature:

```
int f ( int x,
        double y,
```

(Not v

Assu

64 bits, then the AR would look like on the right.



Non-integer procedure arguments

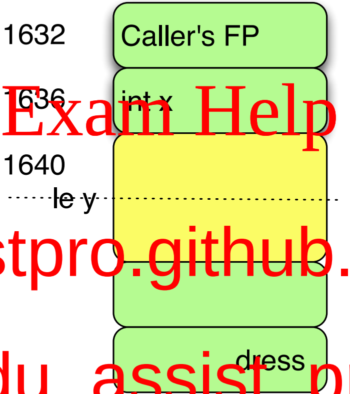
Consider a procedure with the following signature:

```
int f ( int x,
        double y,
```

(Not v

Assu

64 bits, then the AR would look like on the right

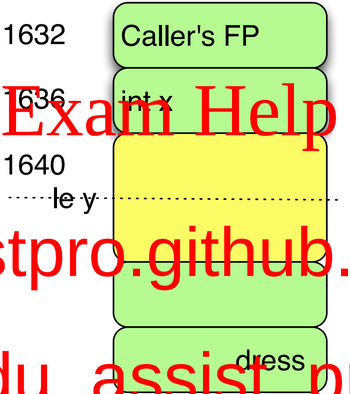


How does the code generator know what size the variables have?

Non-integer procedure arguments

Consider a procedure with the following signature:


```
int f ( int x,
        double y,
```



(Not w

Assu

64 bits, then the AR would look like on the right.

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 [illegible]
proc [illegible]

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

proc

In this case we'd have to move to an approach t return value also in the AR (either for all argum arguments that don't fit in a register – we know a 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)
```



```
    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)
```

```
exit2:
    lw $ra 4($sp)
    addiu $sp $sp 12
    lw $fp 0($sp)
    jr $ra
```

Interesting observations

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

pus

pop

will s

garb

low-level language (C, C++, Rust) don't hav

(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
pus
pop
will s
garb
low-level language (C, C++, Rust) don't hav
(by default).

The source language has recursion. The tar
(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
pus
pop
will s

garb
low-level language (C, C++, Rust) don't hav
(by default).

The source language has recursion. The tar
(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
pus
pop
will s
garb
low-level language (C, C++, Rust) don't hav
(by default).

The source language has recursion. The tar
(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 pus pop will s garb

low-level language (C, C++, Rust) don't hav (by default).

The source language has recursion. The tar (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
                                                k
```



```
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)       // now we load n back from
                    // the sta
```

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
                                              k
```



```
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)     // now we load n back from
                  // the sta
```


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
                                              k
```



```
addiu $sp $sp -4
li $a0 0
lw $t1 4($sp)      // now we load n back from
                   // the sta
```


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,

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,
-

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,
-
- Java's `main` – other languages mig conventions) 'to get the ball rolling'. Thi 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,
-
-

Java's `main` – other languages mig conventions) 'to get the ball rolling'. Thi involves

1. Creating (the caller's side of) an activati

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,
-
-

Java's `main` – other languages mig conventions) 'to get the ball rolling'. Thi involves

1. Creating (the caller's side of) an activati
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,
-
- Java's `main` – other languages mig conventions) 'to get the ball rolling'. Thi involves

1. Creating (the caller's side of) an activati
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
```



```
    ... // 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
```