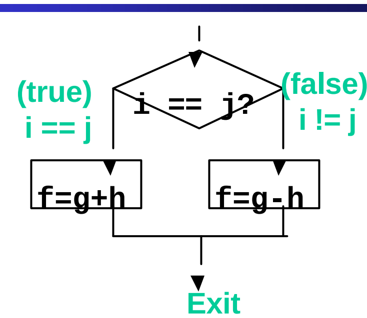
MIPS Instruction Set

(Chapter 3)
EE424 Spring 2003

Compiling C `if` into MIPS

- Compile by hand

```
if (i == j) f=g+h;
else f=g-h;
```



- Use this mapping:

```
f: $s0    g: $s1       h: $s2
i: $s3    j: $s4
```

- Final compiled MIPS code:

```
          beq $s3,$s4,True # branch i==j
          sub $s0,$s1,$s2  # f=g-h(false)
          j   Fin          # go to Fin
True:     add $s0,$s1,$s2  # f=g+h (true)
Fin:
```

Loops

```
do {
        g = g + A[i];
        i = i + j;
  } while (i != h);
```

Register mapping: g: $s1 h: $s2
i: $s3 j: $s4
base of A: $s5

- Final compiled MIPS code:

```
Loop:
              muli $t1,$s3,4           #$t1 =4*I
               add $t1,$t1,$s5     #$t1= @A[i]
               lw  $t1,0($t1)      #$t1=A[i]
               add $s1,$s1,$t1     #g=g+A[i]
               add $s3,$s3,$s4     #i=i+j
               bne $s3,$s2,Loop    # goto Loop
                                   # if i!=h
```


Inequalities in MIPS

- Until now, we've only tested equalities
  (== and != in C).
  General programs need to test < and > as well.

- Create a MIPS Inequality Instruction:
  - "Set on Less Than"
  - Syntax: `slt reg1,reg2,reg3`
  - Meaning:
    ```
    if  (reg2 < reg3)
         reg1 = 1;
         else reg1 = 0;
    ```

- Remark: "set" means "set to 1"
  "reset" means "set to 0".

Inequalities in MIPS (cont'd)

- How to use this?
  Compile by hand:

```
if (g < h) goto Less;
```

- Use the mapping: g: $s0 h: $s1
- Final MIPS code:

```
slt $t0,$s0,$s1   # $t0 = 1 if g<h
bne $t0,$0,Less   # goto Less
                  # if $t0!=0(if (g<h))
```

- Branch if `$t0` != 0 ➔ (g < h)
- Register $0 always contains the value 0 …
  so `bne` and `beq` often use it for comparison
  … after an `slt` instruction!

Inequalities in MIPS (cont'd)

- Now, we can implement <
  … but how do we implement >, <= and >= ?
- We could add 3 more instructions, but:
  - MIPS goal: Simpler is Better
- Can we implement <= in one or more instructions using just `slt` and the branches?
  - What about >?
  - What about >=?
- There are 4 combinations of slt & beq/bneq …

Inequalities in MIPS: <, >, >=, <=

- Here are the 4 combinations of slt & beq/bneq:

<

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
bne $t0,$0,Less # if(g<h) goto Less
```

>

```
slt $t0,$s1,$s0 # $t0 = 1 if g>h
bne $t0,$0,Grtr # if(g>h) goto Grtr
```

>=

```
slt $t0,$s0,$s1 # $t0 = 1 if g<h
beq $t0,$0,Gteq # if(g>=h)goto Gteq
```

<=

```
slt $t0,$s1,$s0 # $t0 = 1 if g>h
beq $t0,$0,Lteq # if(g<=h)goto Lteq
```


Immediate in Inequalities

- There is also an immediate version of `slt` to test against constants: `slti`
  - Helpful in `for` loops

C

```
    if (g >= 1) goto Loop
Loop:    . . .
```

MIPS

```
slti $t0,$s0,1      # $t0 = 1 if
                    # $s0<1 (g<1)
 beq  $t0,$0,Loop   # goto Loop
                    # if $t0==0
                    # (if (g>=1))
```

Unsigned numbers

- There are unsigned inequality instructions:

  `sltu`, `sltiu`

  which set result to 1 or 0 depending on unsigned comparisons

- \$s0 = FFFF $FFFA_{hex}$, \$ s1 = 0000 $FFFA_{hex}$
- What is value of \$t0, \$t1?
  - slt   \$t0, \$s0, \$s1
  - sltu   \$t1, \$s0, \$s1

C Switch (Case) Statement

- Choose among four alternatives depending on whether k has the value 0, 1, 2 or 3.
- Compile this C code:

```
switch (k) {
  case 0: f=i+j; break; /* k=0*/
  case 1: f=g+h; break; /* k=1*/
  case 2: f=g–h; break; /* k=2*/
  case 3: f=i–j; break; /* k=3*/
  }
```

C Switch Statement (cont'd)

- This is complicated, so simplify!
- Rewrite it as a chain of if-else statements (which we already know how to compile)

```
if(k==0) f=i+j;
  else if(k==1) f=g+h;
    else if(k==2) f=g–h;
      else if(k==3) f=i–j;
```

- Use this mapping:

f: $s0	g: $s1	h: $s2
i: $s3	j: $s4	k: $s5

C Switch Statement (Cont'd)

f: $s0	g: $s1	h: $s2
i: $s3	j: $s4	k: $s5

- Final compiled MIPS code:

```
      bne  $s5,$0,L1        # branch k!=0
      add  $s0,$s3,$s4      #k==0 so f=i+j
      j    Exit             # end … so Exit
L1:   addi $t0,$s5,-1       # $t0=k-1
      bne  $t0,$0,L2        # branch k!=1
      add  $s0,$s1,$s2      #k==1 so f=g+h
      j    Exit             # end … so Exit
L2:   addi $t0,$s5,-2       # $t0=k-2
      bne  $t0,$0,L3        # branch k!=2
      sub  $s0,$s1,$s2      #k==2 so f=g-h
      j    Exit             # end … so Exit
L3:   addi $t0,$s5,-3       # $t0=k-3
      bne  $t0,$0,Exit      # branch k!=3
      sub  $s0,$s3,$s4      #k==3 so f=i-j
Exit: …
```

C Switch Statement (cont'd)

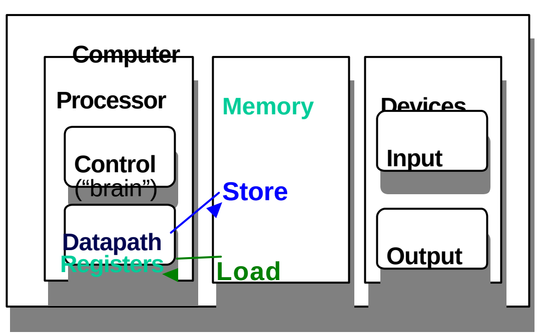
- Sometimes the alternatives of a switch statement can be encoded as
  – A table of addresses (of alternative instruction sequences)
- The program needs only to index the "jump address table" and then jump to the appropriate sequence
- MIPS has a "jump register" instruction `jr`

  It does an unconditional jump to …
  the address specified by the register

Write the corresponding MIPS code!

5 components of any computer

Registers are in the datapath of the processor; if operands are in memory, we must transfer them to the processor to operate on them, And then transfer back to memory when done



These are "data transfer" instructions

Data Transfer: Memory → Reg

- To transfer a word of data
  we need to specify two things:
  - Register: specify this by number (0 - 31)
  - Memory address: more difficult
    - Think of memory as a single one-dimensional array, so we can address it simply by supplying a pointer to a memory address
    - Other times, we want to be able to offset from this pointer

Data Transfer: Memory → Reg (Cont'd)

- To specify a memory address to copy from
  specify two things:
  - A register which contains a pointer to memory
  - A numerical offset (in bytes)

- The desired memory address is …
  the sum of these two values

- Example: `8($t0)`
  - Specifies the byte memory address pointed to by the value in

    `$t0`, plus 8 bytes

Data Transfer: Memory to Reg (3/4)

- Load Instruction Syntax:
  1 2, 3(4)
  - where
    1) operation (instruction) name
    2) register that will receive value
    3) numerical offset in bytes
    4) register containing pointer to memory

  Operation Register/value Offset (Register/pointer)

- Instruction Name:
  - **`lw $t0,8($s0)`**

  (`lw` = Load Word, so load 32 bits or one word from memory at byte address $s0 + 8)

Data Transfer: Memory → Reg (cont'd)

- Example: **`lw $t0,12($s0)`**
  - This instruction will
    - take the pointer in `$s0`
    - add 12 bytes to it, and then
    - load the value from the memory pointed to by this calculated sum into register `$t0`
- Remarks:
  - `$s0` is called the base register
  - 12 is called the offset
- Offset is generally used in accessing elements of array or structure: base register points to beginning of array or structure

Data Transfer: Reg → Memory

- We also want to store the value from a register into memory
- Store instruction syntax is identical to Load
  Instruction Name:

  `sw $t0,8($s0)`

  (**sw** means Store Word)

  32 bits (or one word) are stored to memory at byte address $s0 + 8

- Example: `sw $t0,12($s0)` **$t0 → M[$s0+12]**

  This instruction will take the pointer in **$s0**, add 12 bytes to it, and then store the value from register $t0 into the memory address pointed to by the calculated sum

Pointers v. Values

- Key Concept:
  A register can hold any 32-bit value
  That value can be:
  – a (signed) `int`
  – an `unsigned int`
  – a pointer (memory address).
- If you write `lw $t2,0($t0)`
  Then `$t0` better contain a pointer
- What if you write `add $t2,$t1,$t0`
  Then `$t0` and `$t1` must contain … **values**

Compilation

- What offset in `lw` to select `A[8]` in C?
  4x8=32 to select `A[8]`: byte vs. word
- Compile by hand using registers:
  `g = h + A[8];`

  g: `$s1`
  h: `$s2`
  `$s3`: base address of A

  1st transfer from memory to register:

```
lw    $t0,32($s3)     # $t0 gets A[8]
```

  - Add 32 to `$s3` to select `A[8]`, put into `$t0`

  Next add it to h and place in g

```
add $s1,$s2,$t0      # $s1 = h+A[8]
```


Addressing: Byte vs. word

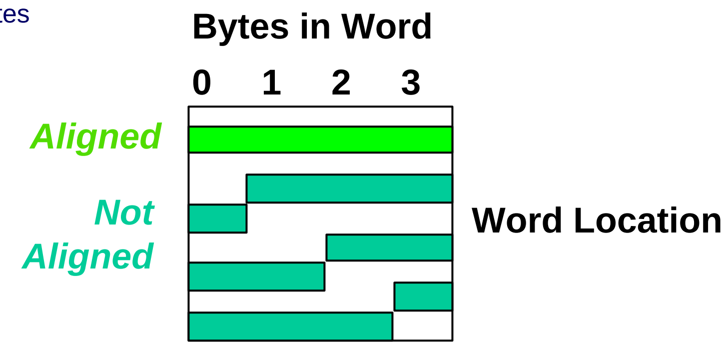
- Every word in memory has an address
  (similar to an index in an array)
- Early computers numbered words like C numbers elements of an array:
  - `Memory[0]`, `Memory[1]`, `Memory[2]`, …

    Called the "**address**" of a word

- Computers needed to access 8-bit bytes
  as well as words (4 bytes/word)
  - For strings: byte data transfers (later)
- Today machines address memory as bytes, hence word addresses differ by 4
  - `Memory[0]`, `Memory[4]`, `Memory[8]`, …

Memory Alignment

- MIPS requires that all words start at addresses that are multiples of 4 bytes



- Called Alignment
  Must fall on address that is multiple of their size.
  - See why when get to caches, pipelining

C functions

```
main() {
  int i,j,k,m;
  i = mult(j,k); ...
  m = mult(i,i); ...
}
/* really dumb mult function */
int mult (int mcand, int mlier){
  int product;
 product = 0;
  while (mlier > 0)  {
   product = product + mcand;
   mlier = mlier -1; }
  return product;
  }
```

What information must compiler/programmer keep track of?

What instructions can accomplish this?

Function Call Bookkeeping

- Registers play a major role in keeping track of information for function calls

- Register conventions:
  - Return address `$ra`
  - Arguments `$a0, $a1, $a2, $a3`
  - Return value `$v0, $v1`
  - Local variables `$s0, $s1, … , $s7`

- The stack is also used.

Function/Procedure Call –Steps (p.132)

1. Place parameters in a place where procedure can access them.
2. Transfer control to procedure
3. Acquire storage resources
4. Perform task
5. Place result value(s) in a place(s) where the calling program can access it (them)
6. Return control to the point of origin

Function/Procedure Call –Steps

1. Place parameters in a place where procedure can access them.

   \$a0 -- \$a3: argument registers

2. Transfer control to procedure

   jal ProcedureAddress (jal: jump-and-link)

   \$ra ← return address (which is PC+4)

   PC ← ProcedureAddress

Function/Procedure Call –Steps

3. Acquire storage resources

   If more register are needed, the stack can be used.

4. Perform task

5. Place result value(s) in a place(s) where the calling program can access it (them)

   \$v0, \$v1: value registers that return values

6. Return control to the point of origin

   jr \$ra

Instruction Support for Functions

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
    return x+y;
}
```

MIPS

```
address
1000 add   $a0,$s0,$zero       # x = a
1004 add   $a1,$s1,$zero       # y = b
1008 addi  $ra,$zero,1016      #$ra=1016
1012 j     sum                 #jump to sum
1016 ...
2000 sum: add $v0,$a0,$a1
2004 jr    $ra              # new instruction
```


Instruction Support for Functions

- Single instruction to jump and save return address: jump and link (`jal`)
- Before:

```
1008 addi $ra,$zero,1016      #$ra=1016
1012 j sum                    #go to sum
```

- After:

```
1012 jal sum        # $ra=1016,go to sum
```

- Why have a `jal`?
  Make the common case fast:
  functions are very common.

Instruction Support for Functions

- Syntax for `jal` (jump and link) is same as for `j` (jump):

```
jal   label
```

- `jal` should really be called `laj` for "link and jump":
  - Step 1 (link): Save address of *next* instruction into $ra (Why next instruction? Why not current one?)
  - Step 2 (jump): Jump to the given label
- Syntax for `jr` (jump register):

```
jr    register
```

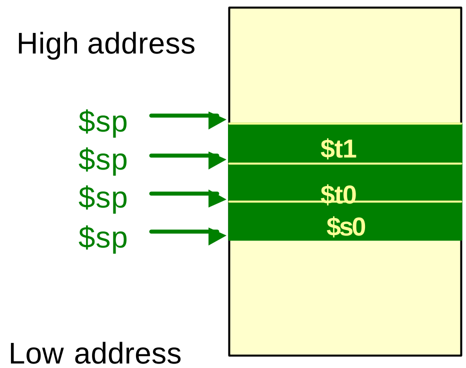
- Instead of providing a label to jump to, the `jr` instruction provides a register which contains an address to jump to
- Only useful if we know exact address to jump to
- Very useful for function calls:
  - `jal` stores return address in register ($ra)
  - `jr` jumps back to that address

Using the stack (p134)

```
int leaf_example (int g, int h, int i, int j)
{
     int f;
     f = (g+h) – (i+j)
     return f;
}
```

Stack (cont'd)

Assume: caller has important data in $s0,$t0, and $t1; the procedure uses these registers.



Stack (cont'd) "PUSH"

Assume: caller has important data in $s0,$t0, and $t1; the procedure uses these registers.

```
sub $sp, $sp, 12     # we make room for 3 registers
sw $t1, 8($sp)       # save reg $t1
sw $t0, 4($sp)       # save reg $t0
sw $s0, 0($sp)       # save reg $s0
```

Stack (cont'd)

g:$a0 h:$a1 i:$a2 j:$a3

f = (g+h)-(i+j)

```
add $t0, $a0, $a1     # t0 ← g+h
add $t1, $a2, $a3     # t1 ← i+j
sub $s0, $t0, $t1     # f=t0-t1

move $v0,$s0          # Return value of f ($v0)
```


Stack (cont'd) "POP"

RESTORE OLD VALUES before returning to caller

```
lw $s0, 0($sp)        # restore reg $s0
lw $t0, 4($sp)        # restore reg $t0
lw $t1, 8($sp)        # restore reg $t1
add $sp, $sp, 12      # adjust stack to delete 3 items

jr  $ra               # jump back to caller
```

MIPS register types

$t0-$t9: temporary registers that are not preserved when a procedure is called.

$s0-$s7: saved registers that must be preserved.

Nested Procedures

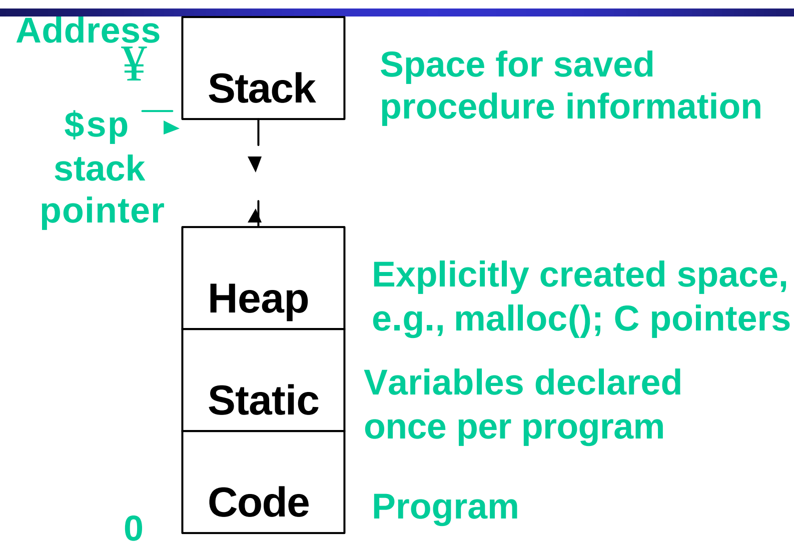
```
int sumSquare(int x, int y) {
     return mult(x,x)+ y;
  }
```

- Something called `sumSquare`, now `sumSquare` is calling `mult`.
- So there's a value in $ra that `sumSquare` wants to jump back to, but this will be overwritten by the call to `mult`.
- Need to save `sumSquare` return address before call to `mult`.

Nested Procedures

- In general, may need to save some other info in addition to $ra.
- When a C program is run, there are 3 important memory areas allocated:
  - Static: Variables declared once per program, cease to exist only after execution completes. E.g., C globals
  - Heap: Variables declared dynamically
  - Stack: Space to be used by procedure during execution; this is where we can save register values

C memory Allocation



Using the Stack

- So we have a register $sp which always points to the last **<u>used space</u>** in the stack
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info
- So, how do we compile this?

```
int sumSquare(int x, int y) {
 return mult(x,x)+ y;
}
```


Using the Stack (cont'd)

•Hand-compile

```
int sumSquare(int x, int y) {
    return mult(x,x)+ y; }
```

```
sumSquare:
"push"  addi $sp,$sp,-8     # space on stack
        sw $ra, 4($sp)      # save ret addr
        sw $a1, 0($sp)      # save y

        add $a1,$a0,$zero   # mult(x,x)
        jal mult            # call mult

        lw $a1, 0($sp)      # restore y
        add $v0,$v0,$a1     # mult()+y
"pop"   lw $ra, 4($sp)      # get ret addr
        addi $sp,$sp,8      # restore stack
        jr $ra
mult: ...
```

Steps for Making a Procedure Call

1) Save necessary values onto stack.
2) Assign argument(s), if any.
3) `jal` call
4) Restore values from stack.

Rules for Procedures

- Called with a `jal` instruction
- Returns with a `jr $ra`
- Accepts up to 4 arguments in $a0, $a1, $a2 and $a3
- Return value is always in $v0 (and if necessary in $v1)
- Must follow register conventions (even in functions that only you will call)! So what are they?

MIPS Registers

The constant 0	$0	$zero
Reserved for Assembler	$1	$at
Return Values	$2-$3	$v0-$v1
Arguments	$4-$7	$a0-$a3
Temporary	$8-$15	$t0-$t7
Saved	$16-$23	$s0-$s7
More Temporary	$24-$25	$t8-$t9
Used by Kernel	$26-27	$k0-$k1
Global Pointer	$28	$gp
Stack Pointer	$29	$sp
Frame Pointer	$30	$fp
Return Address	$31	$ra

(From COD 2nd Ed. p. A-23)

Register Conventions

- Caller: the calling function
- Callee: the function being called

- When callee returns from executing, the caller needs to know which registers may have changed and which are guaranteed to be unchanged

- Register Conventions:
  A set of generally accepted rules as to
  - which registers will be unchanged after a procedure call (`jal`)
  - and which may be changed

Register Conventions (cont'd)

- $0: No Change. Always 0.
- $s0-$s7: No Change.
  Very important!
  That's why they're called saved registers.
  If the callee changes these in any way,
  it must restore the original values before
  returning.
- $sp: No Change.
  The stack pointer must point to the same place
  before and after the `jal` call

  … or else the caller won't be able to restore values from the stack!

Register Conventions (cont'd)

- $ra: Change.
  The `jal` call itself will change this register.
  Caller needs to save on stack if nested call.
- $v0-$v1: Change.
  These are expected to contain
  the new returned values.
- $a0-$a3: Change.
  These are volatile argument registers.
  Caller needs to save if they'll need them after the call.
- $t0-$t9: Change.
  That's why they're called temporary:
  any procedure may change them at any time.
  Caller needs to save if they'll need them afterwards.

Register Conventions (cont'd)

- What do these conventions mean?
  - If function R calls function E, then function R must save any temporary registers that it may be using onto the stack before making a `jal` call.
  - Function E must save any S (saved) registers it intends to use before garbling up their values
  - Remember: Caller/callee need to save only temporary/saved registers they are using, not all registers.
- Note that, if the callee is going to use some s registers, it must:
  - save those s registers on the stack
  - use the registers
  - restore s registers from the stack
  - `jr $ra`
- With the temp registers, the callee doesn't need to save onto the stack.
- Therefore the caller must save those temp registers that it would like to preserve though the call.

Other Registers

- $at: may be used by the assembler at any time; unsafe to use
- $k0-$k1: may be used by the kernel at any time; unsafe to use
- $gp: don't worry about it
- $fp: don't worry about it

- Note: Feel free to read up on $gp and $fp in Appendix A, but you can write perfectly good MIPS code without them.

Remember …

- Functions are called with `jal`, and return with `jr $ra`.

- The stack is your friend:
  Use it to save anything you need.
  Just be sure to leave it the way you found it.

- Register Conventions:
  Each register has a purpose and limits to its usage.
  Learn these and follow them, even if you're writing all the code yourself.

Remember …

- Instructions we know so far
  Arithmetic: `add, addi, sub, addu, addiu, subu`
  Memory: `lw, sw`
  Decision: `beq, bne, slt, slti, sltu, sltiu`
  Unconditional Branches (Jumps):
  `j, jal, jr`

- Registers we know so far
  – All of them!

Example

```
main() {
  int i,j,k,m; /* i-m:$s0-$s3 */
     i = mult(j,k); ...
     m = mult(i,i); ...
}
int mult (int mcand, int mlier){
  int product;
 product = 0;
  while (mlier > 0)  {
   product += mcand;
   mlier -= 1; }
  return product;
  }
```


Example (cont'd)

```
__start:
 add $a0,$s1,$0     # arg0 = j (a0←j)
 add $a1,$s2,$0     # arg1 = k (a0←j)
 jal mult           # call mult
 add $s0,$v0,$0     # i = mult()
 add $a0,$s0,$0     # arg0 = i
 add $a1,$s0,$0     # arg1 = i
 jal mult           # call mult
 add $s3,$v0,$0     # m = mult()
 ...
done
```

Example (cont'd)

- Notes:
  - `main` function ends with `done`, not `jr $ra`, so there's no need to save `$ra` onto stack
  - all variables used in `main` function are saved registers, so there's no need to save these onto stack

Example (cont'd)

```
mult:
     add  $t0,$0,$0    # prod=0
Loop:
     slt  $t1,$0,$a1   # mlr > 0?
     beq  $t1,$0,Fin   # no=>Fin
     add  $t0,$t0,$a0  # prod+=mc
     addi $a1,$a1,-1   # mlr-=1
     j    Loop         # goto Loop
Fin:
     add  $v0,$t0,$0   # $v0=prod
     jr   $ra          # return
```

Example (cont'd)

- Notes:
  - no `jal` calls are made from `mult` and we don't use any saved registers, so we don't need to save anything onto stack
  - temp registers are used for intermediate calculations (could have used s registers, but would have to save the caller's on the stack.)
  - $a1 is modified directly (instead of copying into a temp register) since we are free to change it
  - result is put into $v0 before returning