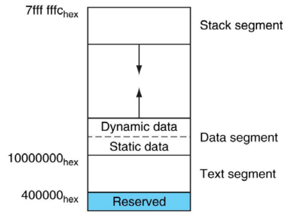
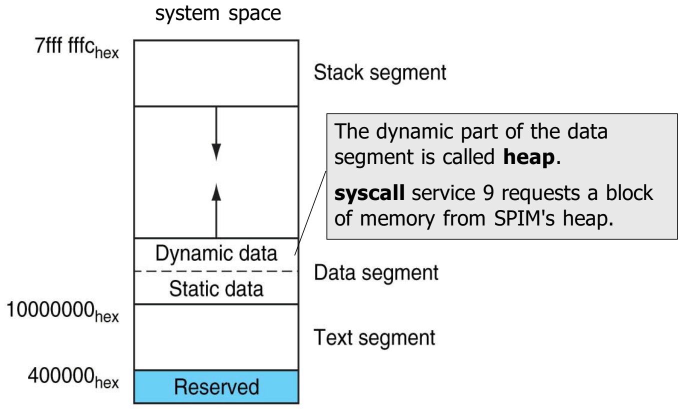
Lecture 4: Memory layout and procedures



Topics

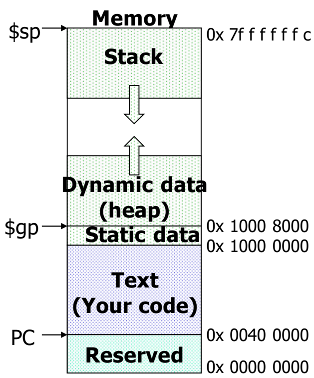
- Memory layout
  - Segments (.data, .text …)
- Memory alignment
  - Mixed data types
- Procedures (PH2 §3.6, PH3 §2.7, PH4 §2.8 or PH5, PH6 §2.8 & HP_AppA P_{22})
  - Using procedures
  - Software support: jal, jr
  - Hardware support for procedures
    - $ra; register conventions
    - Stack and stack conventions

Memory layout [PH2, PH3, A-21; PH4, B-21]



Text segment, data segment

- TEXT SEGMENT
- DATA SEGMENT



Implications of memory layout

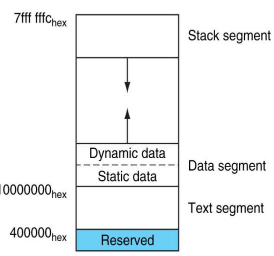
- **Data** segment begins far above the **text** segment
  - load and store instructions cannot use addresses in data segment directly (offset field is 16 bits)
- For example, to load a data item at address 0x1000 8000

```
lui $t0, 0x1000
lw $v0, 0x8000($t0)
```

- the lui instruction has to be repeated for every load and store from/to data segment
- this is done by the assembler

$t0

0001000000000000	0000000000000000

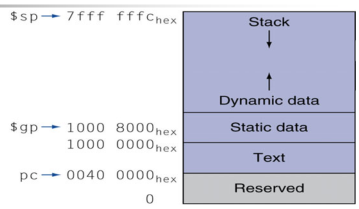


Another convention

```
lui $t0, 0x1000

lw $v0, 0x8000($t0)

lw $v0, 0($gp)
```



- MIPS solution
  - dedicate a register to hold the address of the **data** segment
  - this register is **$gp**, the global pointer register
  - **$gp** contains `0x1000 8000`, it is set by the assembler
  - A single instruction can be used for addressing locations within 2^{16} bytes from the beginning of the data segment (from `0x1000 0000` to `0x1001 0000`)
    - MIPS compilers use this area to store global variables
    - Now we can do (compare this with previous slide):

Mixing data types

- consider the following data declaration:

```
        .data
        .align 0     # turns off auto alignment.
# memory is allocated beginning with the first free byte
str1:  .asciiz "this string has n characters"
abc:   .word 2,4,7,9

# directive .asciiz stores defined string in memory
# and null-terminates it (str1: 28 characters+null)
```

- The string str1 occupies
  - ? bytes
- Thus, words of array abc are NOT ALIGNED
- we have problem as: lw and sw can only operate on aligned words

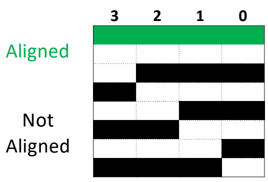
Memory contents without proper alignment

- Memory layout

s	i	h	t	0	
r	t	s	<sp>	4	
<sp>	g	n	i	8	
<sp>	s	a	h	12	
h	c	<sp>	n	16	
c	a	r	a	20	
s	r	e	t	24	
00000000	00000000	00000010	00000000	28	
00000000	00000000	00000100	00000000	32	2
00000000	00000000	00000111	00000000	36	4
00000000	00000000	00001001	00000000	40	7
			00000000	44	9
				48	

Memory alignment, directives

- MIPS requires that all words start at addresses that are multiples of 4
  - – Alignment: objects must fall on address that is multiple of their size



- .align n
  - aligns the next item of data on the 2^n byte boundary
- .align 2
  - aligns the next value on the word boundary
  - word aligned address is divisible by 4
- .align 0
  - turns off automatic alignment until the next .data directive
  - useful if you want to experiment with alignment (RISC and PCSPIM tries to align data automatically)

Memory contents with proper alignment

- Memory layout

s	i	h	t	0
r	t	s	<sp>	4
<sp>	g	n	i	8
<sp>	s	a	h	12
h	c	<sp>	n	16
c	a	r	a	20
s	r	e	t	24
00000000	00000000	00000000	00000000	28
00000000	00000000	00000000	00000010	32
00000000	00000000	00000000	00000100	36
00000000	00000000	00000000	00000111	40
00000000	00000000	00000000	00001001	44
				48

properly aligned data

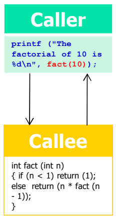
```
        .data
str1:   .asciiz "this
        string has n
        characters"
        .align 2
abc:    .word 2,4,7,9
```


Procedures

- What is a procedure (subroutine, function, method)?

```
main ()
{
      printf ("The factorial of 10 is %d\n", fact(10));
}

int fact (int n)
{
      if (n < 1)
            return (1);
      else
      return (n * fact (n - 1));
}
```



- Why is it used?
  - Large programs are difficult
  - Block structure

Nested and leaf procedures

- A procedure may call other procedures (become a caller)
  - we call these nested procedures
  - if a procedure does not call another procedure we call it a leaf procedure
- Main difference
  - Nested procedures have to preserve the return addresses across the calls (ie. register $ra)
  - Example of leaf procedure

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


A Procedure Call

- How is it implemented?
  - Signature of a procedure

```
int fact (int n)
```

  - Steps required for implementation

1. Place parameters in somewhere (registers?)
2. Transfer control to procedure
3. Acquire storage for procedure
4. Perform procedure's operations
5. Place result in somewhere (registers?) for caller
6. Return to place of call

To speed up execution of procedures registers are used to pass arguments and results;

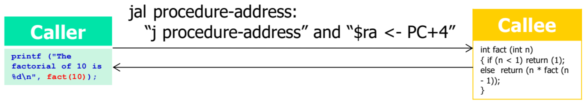
There is only one set of registers; if needed, we spill registers to memory – the STACK

jump-and-link

Register allocation: $a and $v for data transfer

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values(declared variables)	yes
$t8 - $t9	24-25	temporaries	no
$k0, $k1	26, 27	reserved for OS kernel	n.a.
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return address (hardware)	yes

Jump-and-link instruction

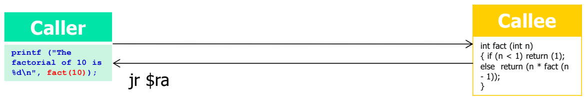


- An instruction to support procedures:

```
jal procedure-address
```

  - jump to procedure-address and simultaneously save the address of the following instruction in $ra (ie. PC + 4)
    - "j procedure-address" and "$ra <- PC+4"
  - storing the return address in $ra forms a link between the procedure and the main program
- Important note
  - the special function of the **$ra** register is enforced by hardware
  - the special function of $a and $v registers is only a convention of usage

Return from procedure



- Use "jump register" instruction

```
jr $ra
```

- This is the last instruction of every procedure
  - we have to use register $ra for return from procedure because of jal instruction
  - but: jr instruction can be used with any other register

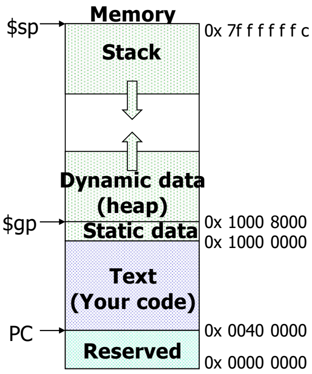
```
int leaf_example(int g, int h, int i, int j)
{
      int f;
      f = (g + h) - (j + i);
      return f;
}
```


Register spilling

- we assign $a0-$a3 and $v0-$v1 to data transfer
- A procedure may need to use other registers
  - there may be more than 4 arguments
  - there may be more than 2 results
- There is only one set of registers
  - The caller uses these registers already
  - A procedure may make no assumptions on the register usage of the caller program (except $a0-$a3, $v0-$v1, and $ra)
- We need to spill registers to memory
  - To do so we use STACK
  - Saving conventions (more explanation later) reduce register spilling -- memory transfer operations are expensive and should be minimised

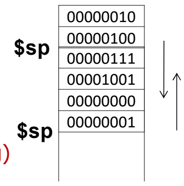
Stack segment

- Working principles
  - last-in-first-out LIFO queue



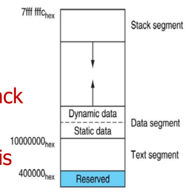
STACK Data Structure

- stack is a last-in-first-out LIFO queue
  - the last item stored on stack is the first item retrieved from stack
  - only the item at the top of the stack is available
- operations on stack
  - **push**: add an item on the top of stack (growing)
  - **pop**: get an item from the top of the stack (shrinking)
- no other operations are allowed
- an ideal stack has no limit on size



Stack implementation in MIPS

- Need an area in memory for the stack
  - the stack starts at a fixed address in memory
  - the total size of the stack is fixed, but is large enough to create an appearance of an ideal stack
- Need to know where the top of the stack is
  - A register **$sp** (stack pointer) is allocated to this function (holds the address of the next free location in the stack)
- The stack always grows from high address in memory to low address in memory



$sp	Stack
subtracting from the pointer e.g. addi $sp,$sp,-12	**Push**: grows the stack
adding to the stack pointer e.g. addi $sp,$sp,12	**Pop**: shrinks the stack

Coding example [1]

- C code

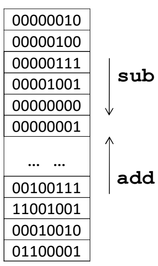
```
int leaf_example(int g, int h, int i, int j)
{       int f;
        f = (g + h) - (j + i);
        return f;
}
```

- MIPS code

jal leaf_example
$v0 = leaf_example($a0, $a1, $a2, $a3)

 # the parameters g, h, i, j correspond to registers $a0, $a1, $a2, $a3
 # variable f corresponds to register $s0

```
prologue
addi $sp,$sp,-12      # make room for 3 items
sw $t0,8($sp)         # save $t0
sw $t1,4($sp)         # save $t1
sw $s0,0($sp)         # save $s0
body
add $t0,$a0,$a1       # $t0 gets g + h
add $t1,$a2,$a3       # $t1 gets i + j
sub $s0,$t0,$t1       # f gets (g+h) - (i+j)
add $v0,$s0,$zero     # return f
epilogue
lw $s0,0($sp)         # restore $s0 for caller
lw $t1,4($sp)         # restore $t1 for caller
lw $t0,8($sp)         # restore $t0 for caller
addi $sp,$sp,12       # shrink stack by 3 items
jr $ra                # jump back to caller
```



Saving conventions

- In the example: 'callee save convention' was used
  - The called procedure saves all registers it will use
- Another possibility: caller save convention
  - The calling program saves all registers it wants preserved
- Yet another possibility
  - A mixed approach with some registers saved by the caller and some by the callee – **both take responsibilities**
  - Memory transfer operations are expensive and should be minimised

```
              # make room for 3 items
leaf_example: sub $sp,$sp,12
              sw $t0,8($sp)   # save $t0
              sw $t1,4($sp)   # save $t1
              sw $s0,0($sp)   # save $s0
```


MIPS convention

- $t0 - $t9 (temporary registers)
  - NOT preserved by the callee on procedure call
  - no assumptions can be made on $t registers usage by the callee
  - **the caller saves and restores ALL $t registers it uses**
- $s0 - $s7 (saved registers)
  - must be preserved on a procedure call, but by whom?
  - no assumptions can be made on $s registers usage by the caller
  - **if used, the callee saves and restores ALL $s registers it uses**
- • aim - reduce register spilling
  - in our code, we only save and restore register $s0, that will reduce 4 memory transfer (sw/lw) instructions
  - if the caller uses $t0 and $t1, the caller has to save and restore them

Coding example [2]

- C code: nested procedures

```
int nested_example (int g, int h, int i, int j)
{
        int f;
        f = sqrt((g + h) - (j + i));
        f = f + 2;
        return f;
}
```

- the parameters g, h, i, j correspond to registers $a0, $a1, $a2, $a3
- variable f corresponds to register $s0
- **sqrt** is a library procedure to calculate square root

Coding example [2]

- MIPS code

jal nested_example
 # $v0 = nested_example($a0, $a1, $a2, $a3)

 # the parameters g, h, i, j correspond to registers $a0, $a1, $a2, $a3
 # variable f corresponds to register $s0

```
prologue
addi $sp,$sp,-8       # make room for 2 items
sw $ra,4($sp)         # save return address
sw $s0,0($sp)         # save $s0
body
add $t0,$a0,$a1       # $t0 gets g + h
add $t1,$a2,$a3       # $t1 gets i + j
sub $t3,$t0,$t1       # $t3 gets (g+h) - (i+j)
add $a0,$t3,$zero     # argument for sqrt
jal sqrt              # call sqrt procedure
add $s0,$v0,$zero     # save result in f
addi $s0,$s0,2        # f gets f+2
add $v0,$s0,$zero     # return f
epilogue
lw $s0,0($sp)         # restore $s0 for caller
lw $ra,4($sp)         # restore $ra
addi $sp,$sp,8        # shrink stack by 2 items
jr $ra                # jump back to caller
```

Stack discipline

See HP_AppA.pdf, A-25

- callee NEVER writes to addresses greater than $sp
  - as illustrated, the area above the caller stack pointer
  - the contents of the stack above stack pointer is preserved
  - the contents of the stack below stack pointer is NOT preserved
- callee ALWAYS adds to $sp exactly the same value it subtracted from $sp
  - the value of $sp is therefore preserved
- if the above two rules are obeyed
  - after the call the caller will find the values it deposited on the stack before the call
- The stack discipline is enforced by convention not hardware

More on stack usage

- 4 registers only reserved for arguments $a0 - $a3
- by MIPS convention
  - additional parameters placed on stack above the frame pointer
  - this is done by the caller
  - these arguments are accessed by the callee using fixed offset from the frame pointer
- 2 registers reserved for return values $v0 - $v1
  - most high level languages only allow one return value
  - there is no convention for more than two return values

More on stack usage

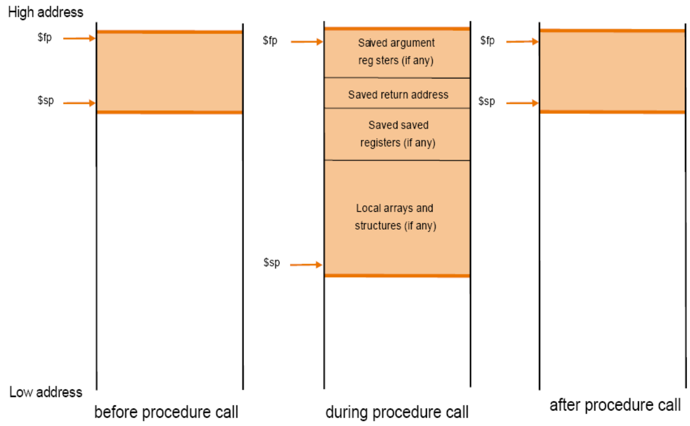
- the stack may also be used to store the local procedure variables
  - simple variables which do not fit into registers
  - local arrays and structures
- procedure frame (activation record)
  - the fragment of the stack containing saved argument registers, saved return address, saved caller registers, local arrays and structures
- MIPS allocates a register $fp to point to the beginning of the frame (frame pointer)
  - this makes finding the items on the stack easy
  - we use $sp for this in lab examples

Frame pointer

See HP_AppA.pdf, A-25

- frame pointer has to be preserved across procedure calls
  - it is specific to procedure activation
  - once set, it does not change during procedure execution
- stack pointer may change during the procedure execution
  - unlike in our examples so far
- frame pointer is a fixed base within the procedure
  - any register saved in the frame has a fixed offset from the $fp
  - the procedure is easier to write and understand
- and again
  - all this is only a convention, not enforced in hardware

Local Data on the Stack



Revision quiz

- MIPS aligns the next item of data on the word boundary using:
  1) `.align 2` 2) `.align 0` 3) `.align 4`
- By conventions, is the usage of registers stated in the following correct?
  "Registers $s0 - $s7 should be saved first by the caller procedure before using them."
- Which of the following can correctly allocate 3 words in the stack?
  1) `subi $sp,$sp,12`
  2) `sub $sp,$sp,12`
  3) `addi $sp,$sp,12`
  2) `add $sp,$sp,12`

Recommended readings

General Data	UnitOutline | LearningGuide | Teaching Schedule | Aligning Assessments |
Extra Materials	ascii_chart.pdf | bias_representation.pdf | HP_AppA.pdf | instruction decoding.pdf | masking help.pdf | PCSpim.pdf | PCSpim Portable Version | Library materials

PH6, §2.8, P102-P112: Procedure calling
PH5, §2.8, P96-P106: Procedure calling
PH4, §2.8, P112-P122: Procedure calling

HP_AppA.pdf -> A-22: Procedure calling

HP_AppA.pdf -> A-24: MIPS registers

HP_AppA.pdf -> A-25: Stack frame

Text readings are listed in Teaching Schedule and Learning Guide

PH6 (PH5 & PH4 also suitable): check whether eBook available on library site

PH6: companion materials (e.g. online sections for further readings)
https://www.elsevier.com/books-and-journals/book-companion/9780128201091

PH5: companion materials (e.g. online sections for further readings)
http://booksite.elsevier.com/9780124077263/?ISBN=9780124077263