CS47 - Lecture 09

Kaushik Patra
(kaushik.patra@sjsu.edu)

1

- Topics
  - Memory Model
  - Stack Operation
  - Global / Local Variable

© 2014 - All rights are reserved by Kaushik Patra

Memory Model ...

2

© 2014 - All rights are reserved by Kaushik Patra

Data Alignment in Memory

Observe how this variables are placed in the memory and their addresses.

```
# Program exp11.asm
.include "../cs47_macro.asm"

.data
var_a: .byte 0x10
var_b: .half 0x3210
var_c: .byte 0x20
var_d: .word 0x76543210

.text
.globl main
main:
    # Variables are read into reg from mem
    lb  $s0, var_a # R[s0] = M[var_a](7:0)
    lh  $s1, var_b # R[s1] = M[var_b](15:0)
    lb  $s2, var_c # R[s2] = M[var_c](7:0)
    lw  $s3, var_d # R[s3] = M[var_d](31:0)

    # System exit
    exit
```

Note that these are pseudo instructions since MIPS supports only register relative addressing.

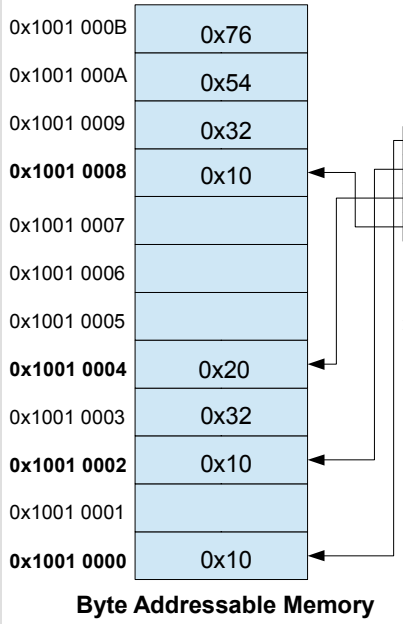
3

- Write this program on MARS exp11.asm and assemble it. After assemble is done of this program, observe the memory layout from starting address 0x10010000 in MARS. Also note the starting address of each of the data label (equivalent to variable in high level programming).

- Once this program is executed, registers s0,s1, s2 and s3 will be holding values of 0x10, 0x3210, 0x20, 0x76543210.

© 2014 - All rights are reserved by Kaushik Patra

Data Alignment in Memory



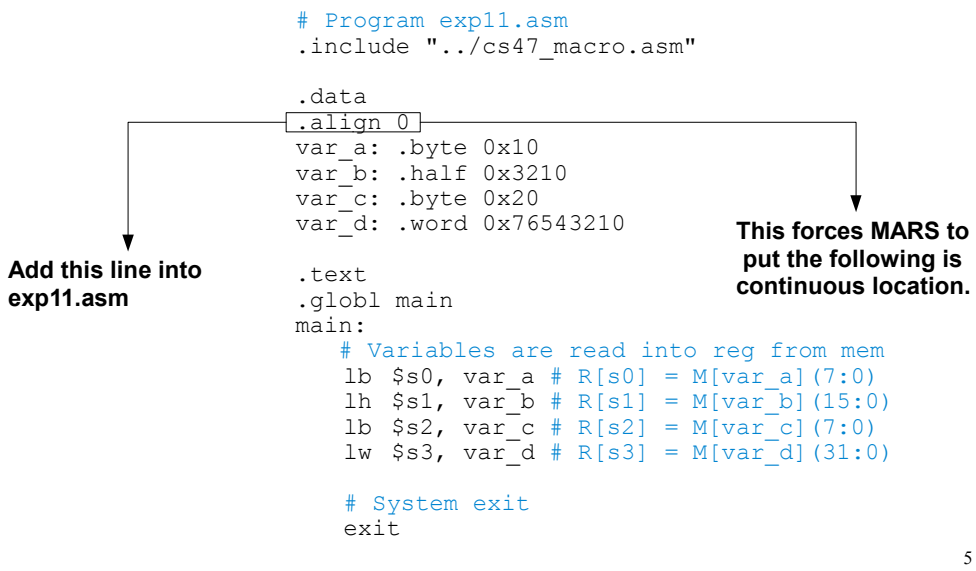

```
# Program exp11.asm
.include "../cs47_macro.asm"

.data
var_a: .byte 0x10
var_b: .half 0x3210
var_c: .byte 0x20
var_d: .word 0x76543210

.text
.globl main
main:
   # Variables are read into reg from mem
   lb $s0, var_a # R[s0] = M[var_a](7:0)
   lh $s1, var_b # R[s1] = M[var_b](15:0)
   lb $s2, var_c # R[s2] = M[var_c](7:0)
   lw $s3, var_d # R[s3] = M[var_d](31:0)

   # System exit
   exit
```

4

- By inspecting into the MARS data segment at 0x10010000, it can be observed that the variables are not placed in contiguous memory location. Why? What if we force MARS to put them in contiguous location?

© 2014 - All rights are reserved by Kaushik Patra

Data Alignment in Memory



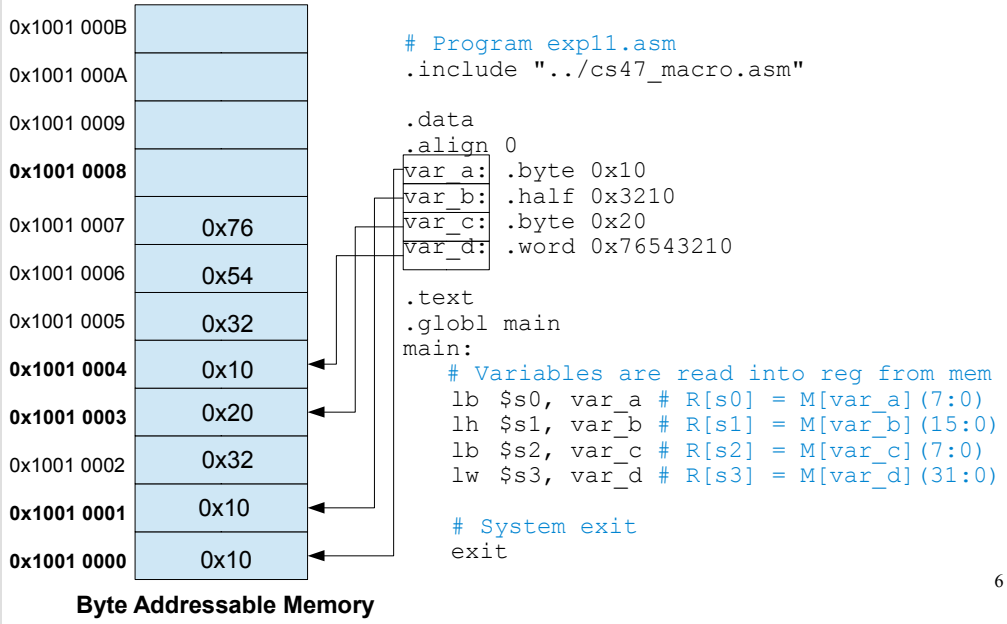

```
# Program exp11.asm
.include "../cs47_macro.asm"

.data
.align 0
var_a: .byte 0x10
var_b: .half 0x3210
var_c: .byte 0x20
var_d: .word 0x76543210

.text
.globl main
main:
    # Variables are read into reg from mem
    lb  $s0, var_a # R[s0] = M[var_a](7:0)
    lh  $s1, var_b # R[s1] = M[var_b](15:0)
    lb  $s2, var_c # R[s2] = M[var_c](7:0)
    lw  $s3, var_d # R[s3] = M[var_d](31:0)

    # System exit
    exit
```

5

- Add the .align directive before starting the data segment variables.
- This '.align n' directive force assembler to put successive datum (meaning chunk of data can be referred by single data label) in 2^n address value boundary. This means that the address will be completely divisible by 2^n , thus the remainder value will always be 0. This concept is called alignment of data. Therefore setting the alignment to 0 means assembler needs to place the successive datum (as defined in the program's .data section) into any address without any constraints.
- Likewise, setting alignment to 1 means the successive datum must be placed in an address completely divisible by 2. An alignment of 2 means the address should be completely divisible by 4. An alignment of 3 means 3 means the address should be completely divisible by 8.

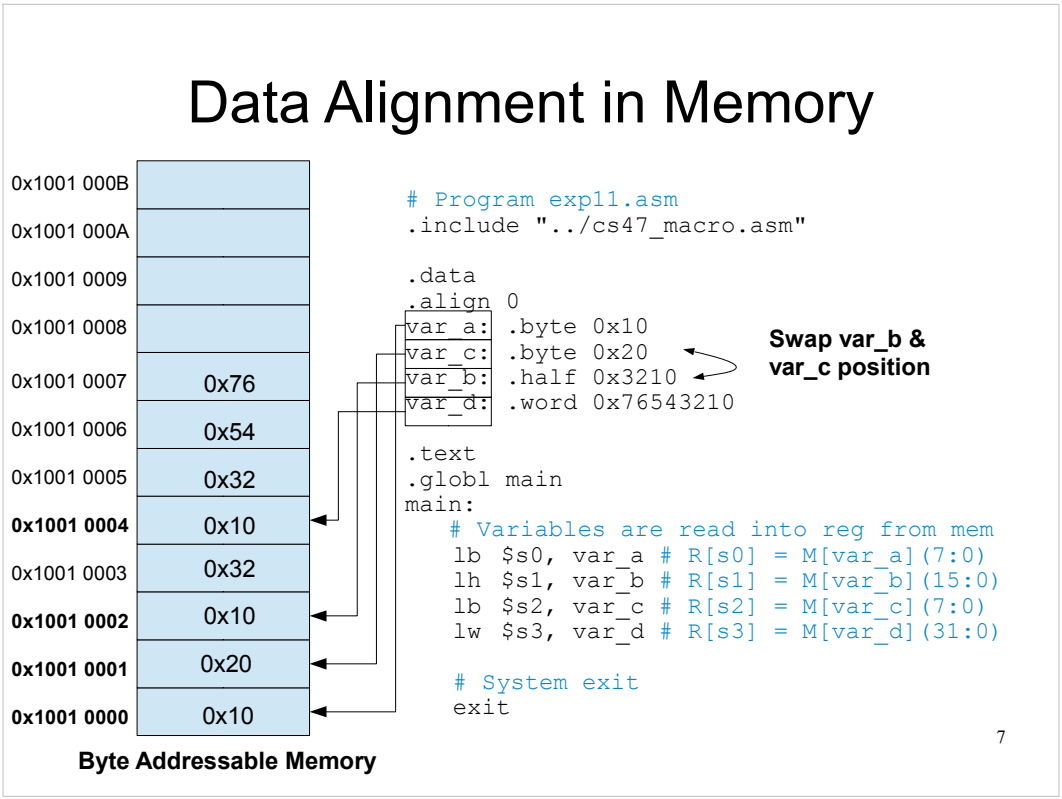
© 2014 - All rights are reserved by Kaushik Patra

Data Alignment in Memory



- After the alignment is forced to 0 (or no alignment) the memory footprint seems very compact (no holes like the previous example).
- However, if we try to run this program, we'll get run time 'alignment' exception while accessing the half word (var_b).
- MIPS has constraint on which data address it can access (for load or store) for each datum type.
  - Single byte type can be accessed at any address (i.e. alignment of 0)
  - Half word type can be accessed only from address completely divisible by 2 (i.e. alignment of 1)
  - Word type can be accessed only from address completely divisible by 4 (i.e. alignment of 2)
- If any of the above access rule is violated run time exception is generated (like in the case of this program – the half word var_b was accessed from address 0x10010001 which is clearly not completely divisible by 2).

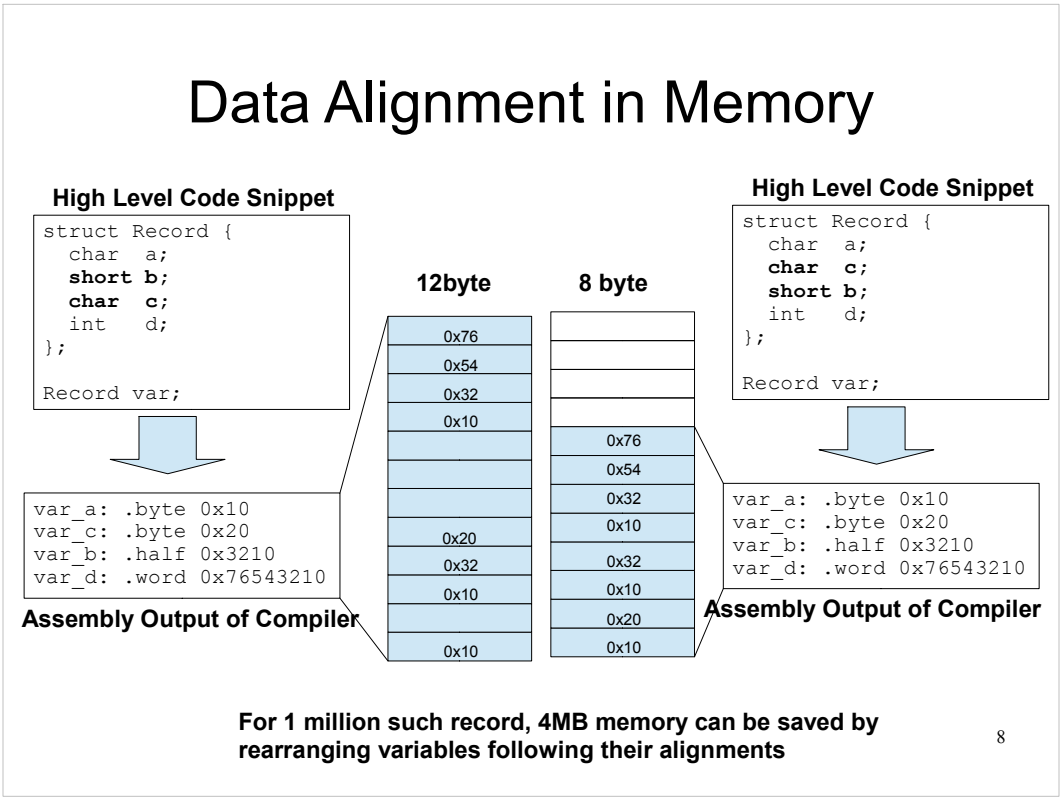
© 2014 - All rights are reserved by Kaushik Patra



- After swapping the position of var_b (half word) and var_c(byte) the run time exception is gone. This is because now the half word access is done from location 0x10010002 which is completely divisible by 2. Therefore it does not violate MIPS data access protocol.

- Assembler tries its best for not to violated data alignment protocol unless overridden by programmers. The very first revision of this experiment in fact did not have any data alignment issues. However, that automatic layout was causing holes in the overall memory layout (the unassigned part in the memory would not be assigned for any other variables since this assignments are one pass – no complicated algorithm is present in the assembler for automatic re-shuffling the user data layout).

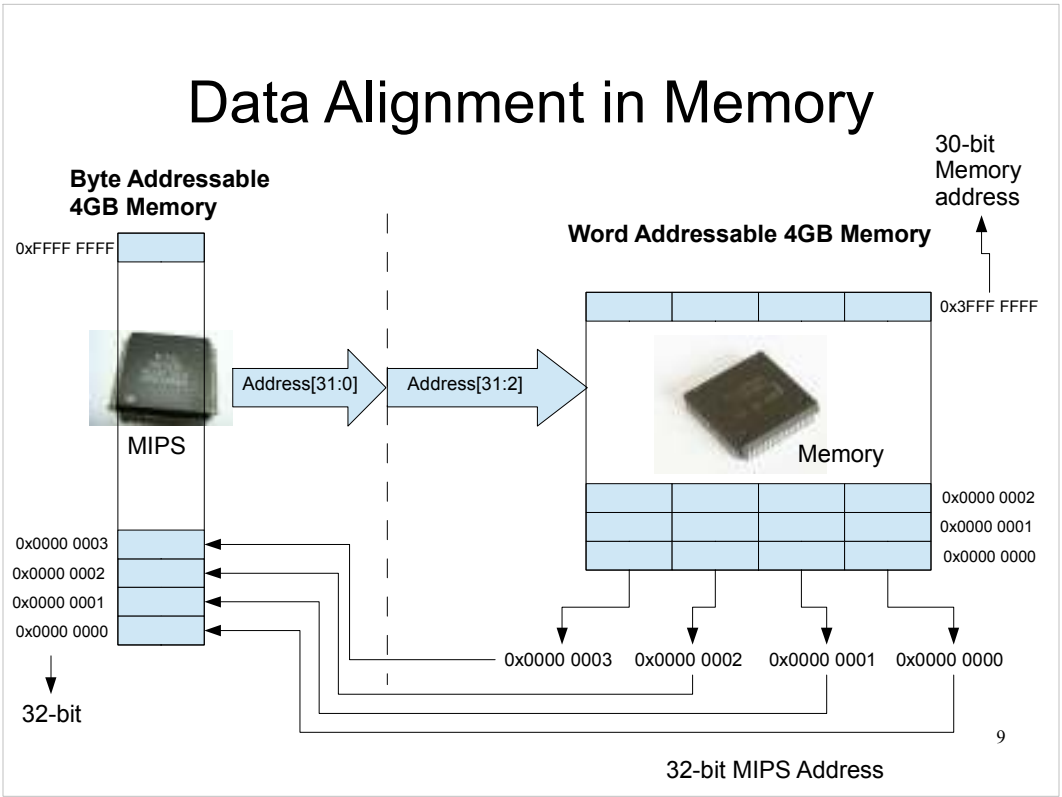
© 2014 - All rights are reserved by Kaushik Patra



- A compiler produces the data memory layout from the variable definition. A structure (class members) are decomposed into uniquely identifiable individual variables (datum) and layed out in data memory. For example any high level access to var.a in this example will cause access to datum var_a at assembly level.

- It important to order the sequence in variable of different sizes in a structure or class (in high level language) to compact the foot print of the structure. The alignment conception can essentially be utilized in ordering the variables to keep the size minimum. Modern compiler tries to rearrange the individual fields/members in a structure/class as much as possible to compact the memory footprint. However, for complex structure (for example structure containing another structure) manual inspection is needed to re-design the structure/class definition.

© 2014 - All rights are reserved by Kaushik Patra

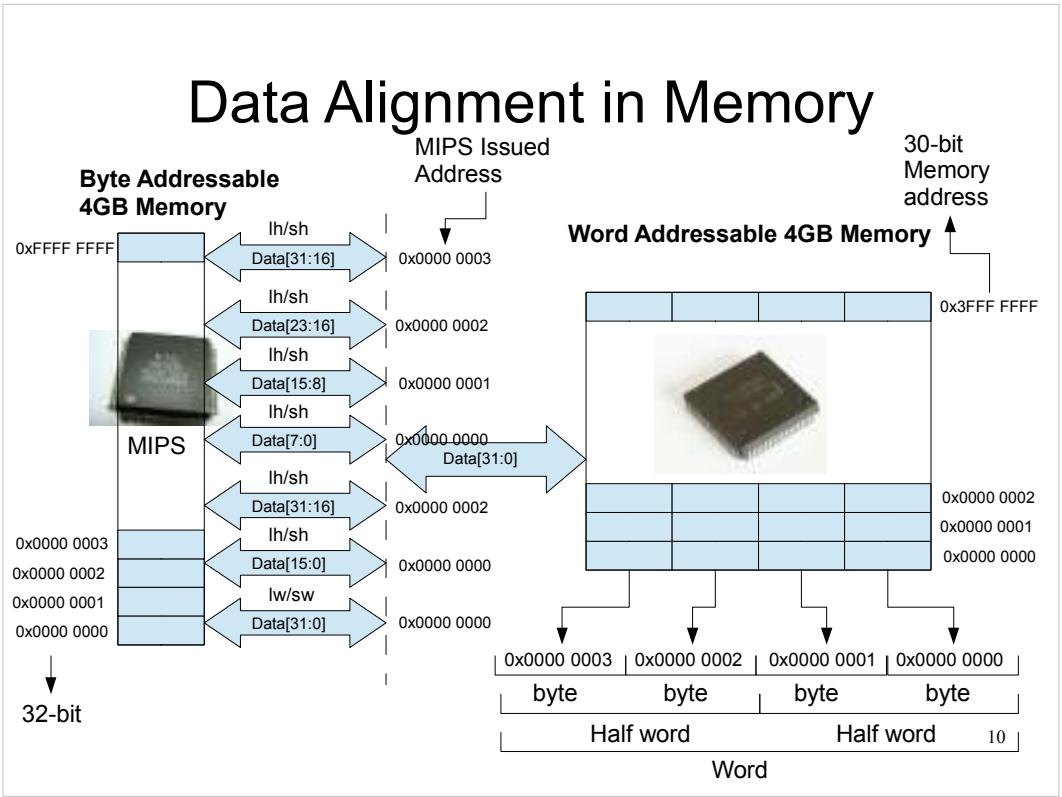


- Though from processor perspective the memory is byte addressable, the memory devices are word addressable. Therefore, each unique address issued to memory device will point to a complete address. This also leads to the fact that 4GB memory device needs only 30-bit address line to point to single word datum. Whereas, processor needs to issue 32-bit address to support byte addressable memory model.

- To map processor issued address into memory address, 2 bits from LSB of the processor issued address is dropped to make a 30-bit address definition for the memory device. This means byte addresses 0x0000 0000, 0x0000 0001, 0x0000 0002 and 0x0000 0003 will be mapped to single 30-bit address 0x0000 0000.

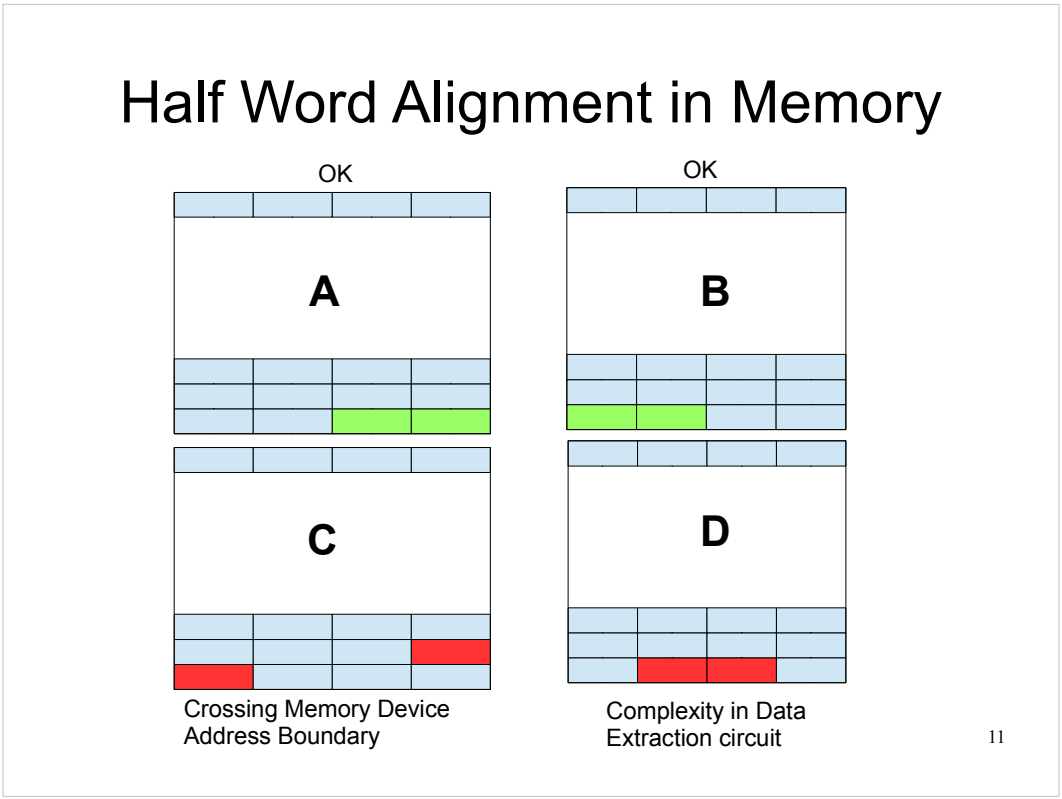
- Conceptually each byte position of the word can be mapped to individual 32-bit byte addresses issued by processor by appending 2-bit byte index in the word to 30-bit memory address. For example the 2-bit byte index in a word for the lowest bytes position is '00' and that for the highest byte position is '11'.

© 2014 - All rights are reserved by Kaushik Patra



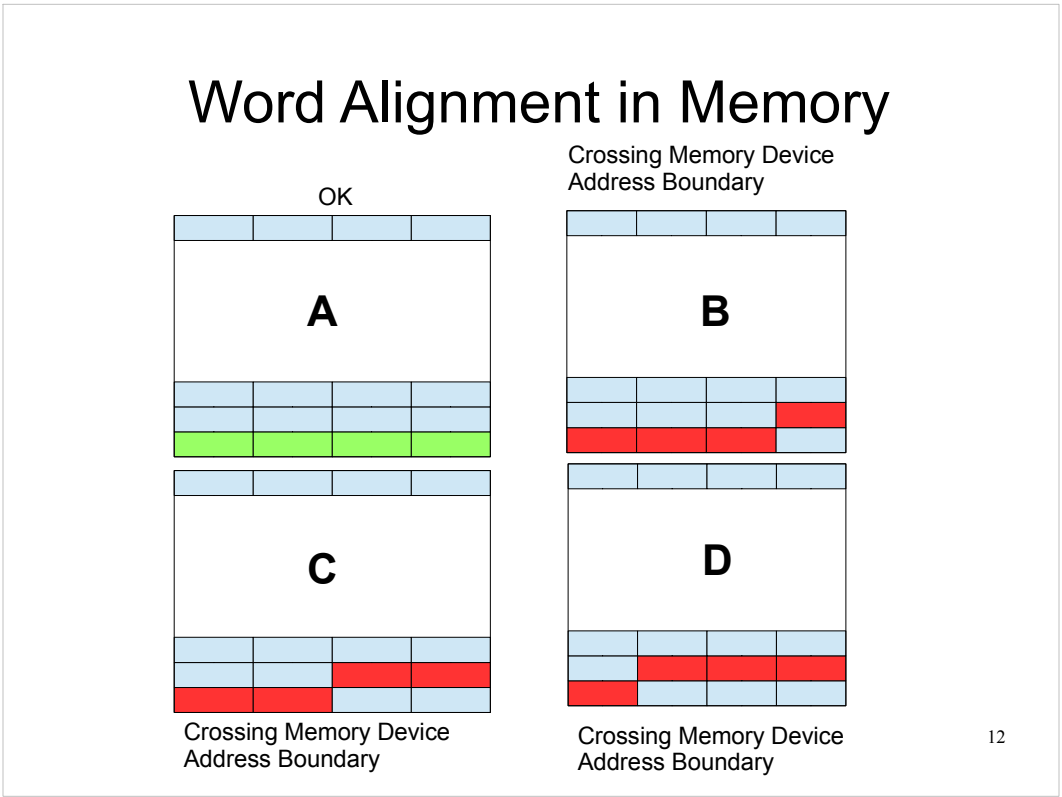
- The memory returns / takes-in 32-bit data for loading / storing. At processor side during byte loading, 32-bit incoming data is filtered to get corresponding byte value for the given byte address. Depending on the byte address being mapped to 1st, 2nd , 3rd, or 4th byte index in the word, corresponding byte value is consumed within processor (with proper sign or zero extension depending on signed or unsigned loading operation). Similarly for half word loading, depending on the half word address, upper or lower half word is taken into processor with sign or zero extension. For loading word, there is no need for any filtering, the complete word from memory is taken into the processor.

© 2014 - All rights are reserved by Kaushik Patra



- MIPS supports alignment for half word as configuration A or B. Therefore MIPS address for half word must be completely divisible by 2. Configuration C is not supported because if a half word exits across memory address boundary then it needs two memory access to completely construct the required half word. Two memory accesses will degrade performance of the program. Configuration D (i.e. the half word is placed in middle of the word – half word address being non-divisible by 2) is not allowed because of complexity will increase in data extraction circuit that extract half word from a full word.

© 2014 - All rights are reserved by Kaushik Patra



- For word access only configuration A supported. Every other configuration (B,C, and D) will cause two memory access which is inefficient. Therefore the MIPS issued 32-bit word address should always be completely divisible by 4. Only with that condition, MIPS issued address can be mapped to an word address of the memory.

- MIPS produce run time exception for any unaligned data access. If this check for alignment was not there, data access would be unreliable. For example, if one is trying to write an word in 0x0000 00002 address, it will override memory location 0x0000 0000 (remember 32-bit 0x00000 0002 will be mapped to 30-bit 0x0000 0000) though the intention was to override the upper half of memory location 0x0000 0000 and lower half of the memory location 0x0000 0001.

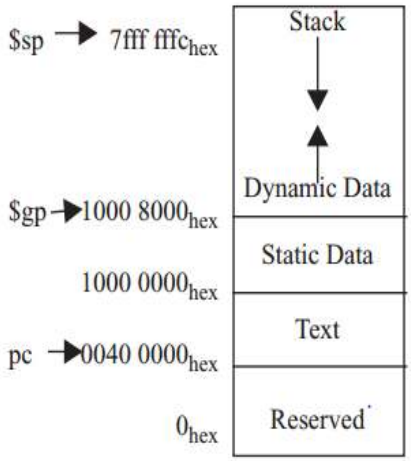
© 2014 - All rights are reserved by Kaushik Patra

Stack Operation ...

13

© 2014 - All rights are reserved by Kaushik Patra

Stack Space in MIPS



- Stack is word accessible space.
- It starts from 0x7ffffffc to be aligned for word storage.
- Stack needs Push and Pop operation.
  - MIPS does not provide native push and pop operation.
  - It is done by manipulating $sp directly.
- Push operation should store data into address pointed by $sp and decrease $sp by 4.
- Pop operation should first increase $sp by 4 and then read word data from that location.

14

- Stack operations are made around word to avoid alignment issues that could have been the case in supporting half word or byte datum push and pop. Supporting just the word datum push/pop it makes the operation implementation uniform and simple.

- Since the max stack address is 0x7fff ffff, the start address of stack if 0x7fff fffc to accommodate complete word withing 0x7fff fffc to 0x7fff ffff (4-bytes). This also confirms the data alignment requirement for word in MIPS.

- Some processor provides the push and pop native operation, but MIPS provides only access to manipulate the stack pointer. User program must take care of the stack space very carefully.

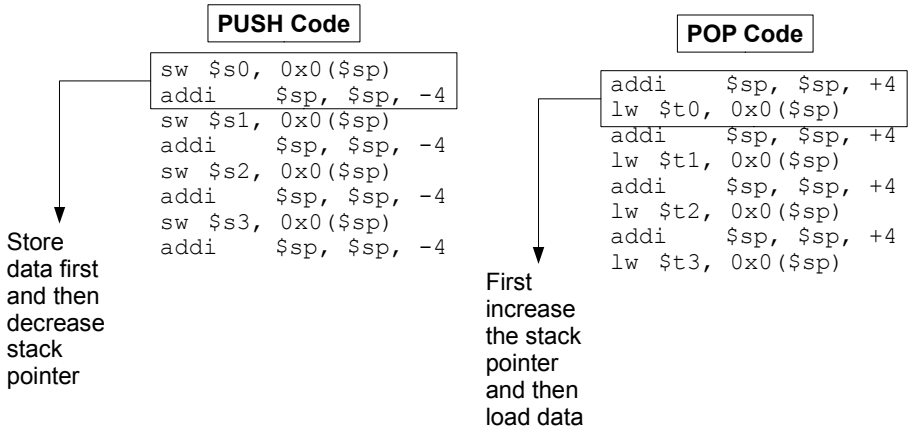
- The push operation always first write at the location pointed by $sp and then decrease $sp value by 4 (remember stack grows lower direction) to accommodate next push operation.

- Since any push operation always progress the $sp, pop operation must first regress the $sp by 4 (subtract 4 from $sp first) and then read the word datum from the location pointed by the latest $sp value.

© 2014 - All rights are reserved by Kaushik Patra

Stack Operations

- Copy exp11.asm to exp12.asm and add the following PUSH and POP code in sequence before the exit.



15

- Once assembled, in MARS load the data segment pointed by stack pointer. Also put break point at first 'sw' operation in push sequence. Once execution comes to this point, step through the program to observe stack segment data changes and $sp value changes.

- Note that the pop operation really does not clear the data space occupied by the datum popped. It may the address space writable for next push.

- Write macro for push and pop operations which takes a register name as argument to save the pop data or supply the data for push. Re-write the program using those macro.

© 2014 - All rights are reserved by Kaushik Patra

Global & Local Variables ...

16

© 2014 - All rights are reserved by Kaushik Patra

Global and Local Variables

Exp14-main.c

```
#include <stdio.h>

int var_a;

void main() {
   printf("Enter an integer : ");
   scanf("%d",&var_a);
   print_gva();
   printf("---> Control is \
            back to main\n");
   return;
}
```

Exp14-main.asm

```
.include "../cs47_macro.asm"

.extern glob_var_a 4

.data
msg1: .asciiz "Enter an integer : "
msg2: .asciiz "---> Control back to main\n"

.text
.globl main
main:
     print_str(msg1)
     read_int($t0)
     sw $t0, glob_var_a
     jal print_gva
L1:  print_str(msg2)

     exit
```

17

- A top level C program like Exp14-main.c can be translated into assembly similar to Exp14.-main.asm.

- Create a separate directory for this experiment and create two assembly files Exp14-main.asm and Exp14-pgva.asm (source code in next page). Also turn on 'compile all source in dir' from the MARS 'setting' menu.

- Global variable var_a is translated into '.extern global_var_a 4'. This directive tells assembler to reserve 4 byte space from available address in global space (starting from 0x1000 0000 at power up of the processor). This space can also be referred from other assembly program with the same name. For example, this label 'global_var_a' is referred from the Exp14-pgva.asm, though it is defined in Exp14-main.asm program.

- All the local variable and constants are placed in the '.data' space (starting from 0x1001 0000 in MARS)

© 2014 - All rights are reserved by Kaushik Patra

Global and Local Variables

Exp14-pgva.c

```
#include <stdio.h>

extern int var_a;

void print_gva() {
   printf("    ----> Control \
             at print_gva\n");
   printf("Global var \
           value is ");
   printf("%d",var_a);
   printf("\n");
   return;
}
```

Exp14-pgva.asm

```
.include "../cs47_macro.asm"

.data
msg1: .asciiz "Global var value is "
strCR: .asciiz "\n"
msg2: .asciiz "    ----> Control at print_gva\n"

.text
.globl print_gva
print_gva:
     print_str(msg2)
     print_str(msg1)
     lw $t1, glob_var_a
     print_reg_int($t1)
     print_str(strCR)
     jr $ra
```

18

- 'jal' is similar to calling a procedure at high level program (this will be discussed in details later). The register 'ra' contains the return point of the caller, in this case label 'L1'.

- Assemble and run this program – observe the control transfer between main and the print_gva. Put a break point on 'jr $ra' and observe the $ra value. It should be same as address for 'L1'. One step execution from this should change the PC value to the address of L1.

© 2014 - All rights are reserved by Kaushik Patra

CS47 - Lecture 09

Kaushik Patra
(kaushik.patra@sjsu.edu)

19

- Topics
  - Memory Model
  - Stack Operation
  - Global / Local Variable

© 2014 - All rights are reserved by Kaushik Patra