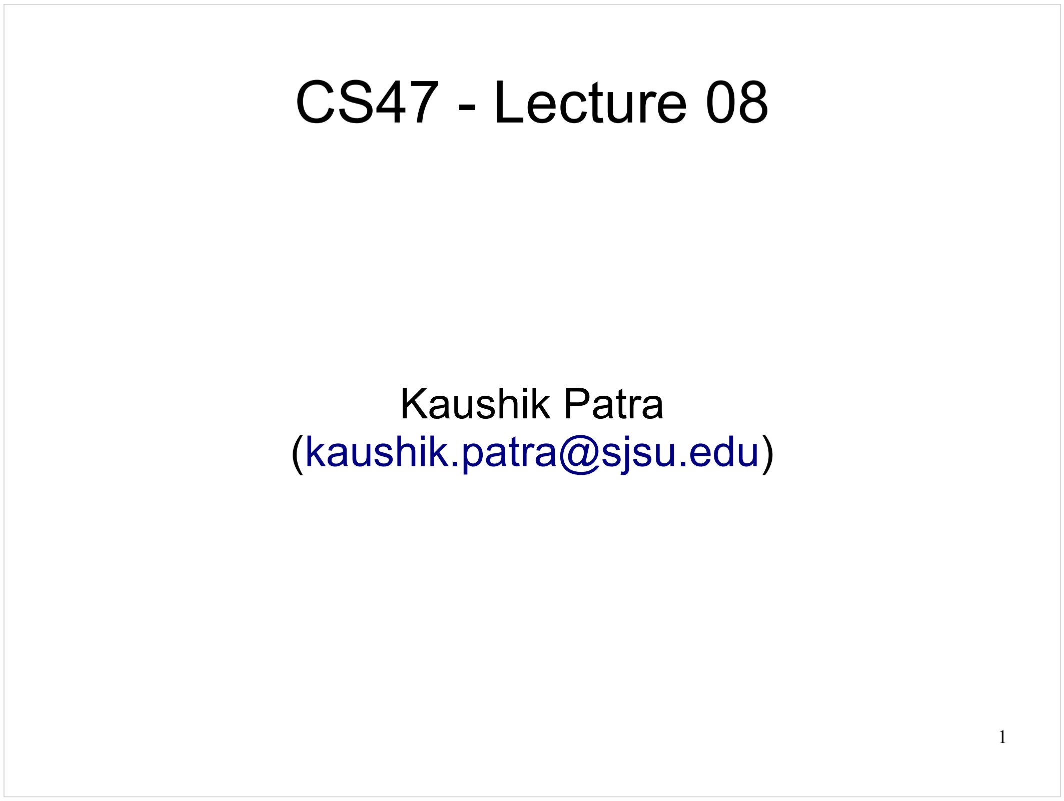
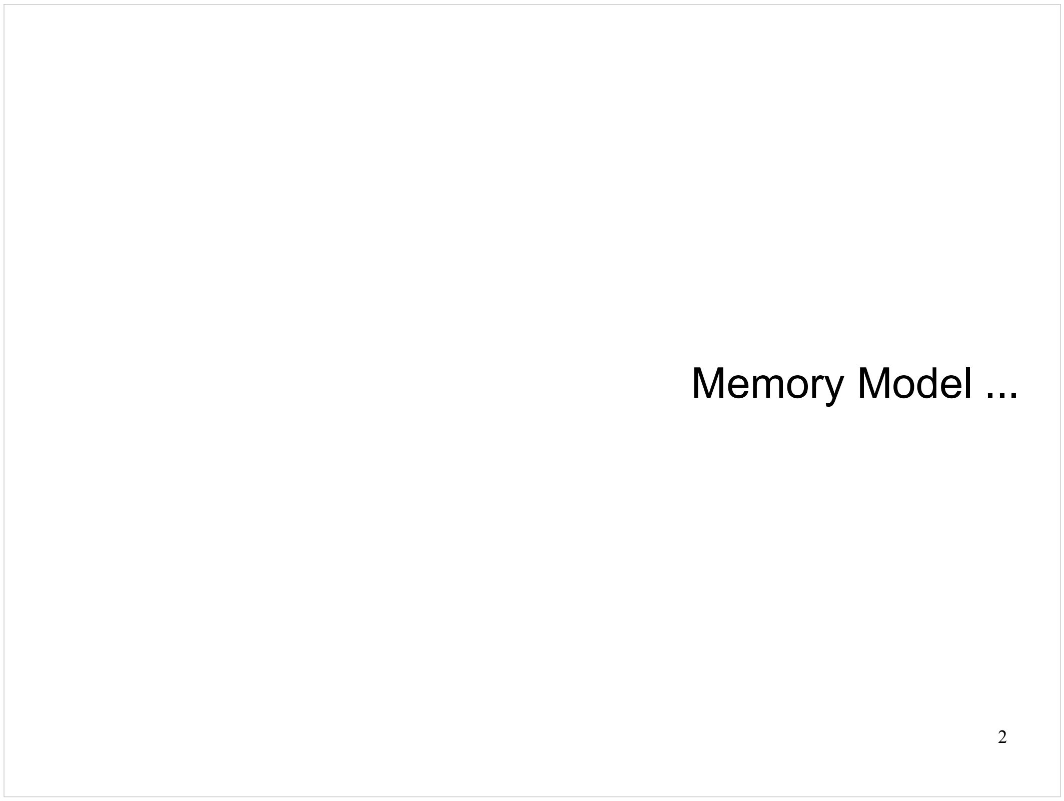


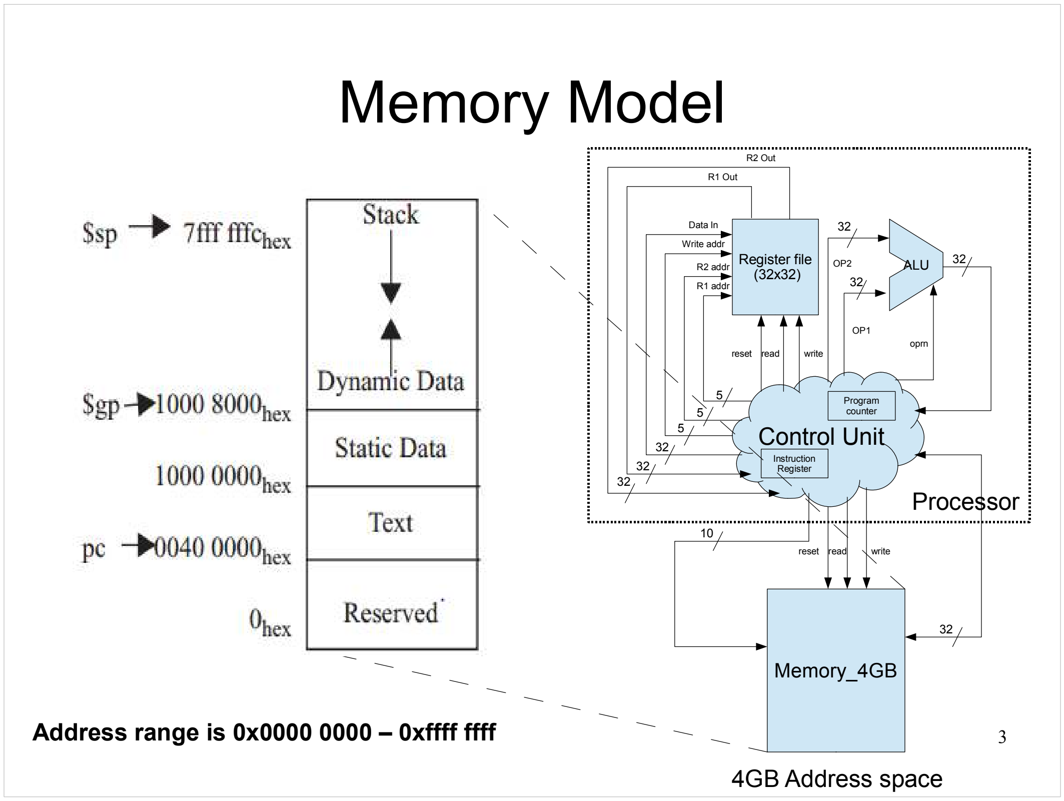
CS47 - Lecture 08

Kaushik Patra
(kaushik.patra@sjsu.edu)

- Topics
    - Memory Model



Memory Model ...



Memory Model

Address range is 0x0000 0000 – 0xffff ffff

3

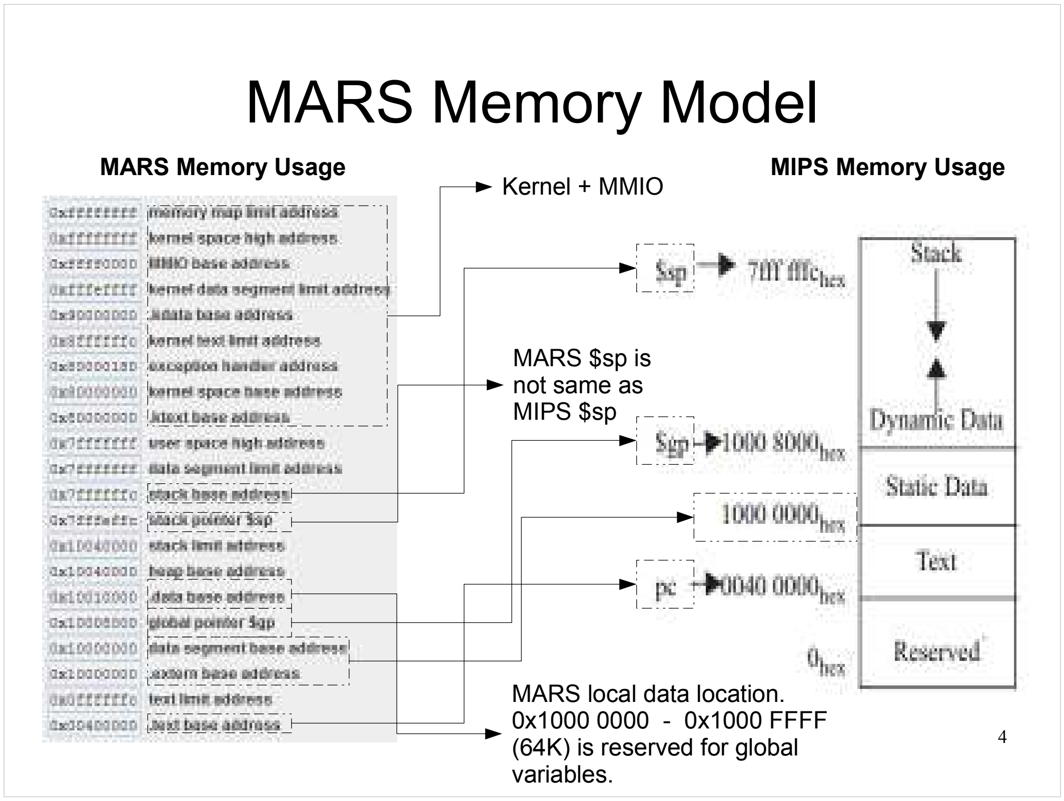
- Ideally user program should be able to write (at least from low level programming without any operating system intervention) the entire 32-bit memory address space (which is 4GB space). However, software system follows some convention and that leads into partition the space into multiple sections for different special purpose. Any system software tries to enforces these boundaries at high level programming.

- Space from 0x0000 0000 to 0x003F FFFF is reserved (4K) is reserved space – mainly for the supporting virtual memories (this part is taught in CS147). User program should not write anything into this space.

- Program execution code is called 'text'. An address space from 0x0040 0000 to 0x0FFF FFFF (252K) is reserved for storing the program's machine code. Sophisticated system software technique has been developed (like dynamically linked library or *.dll / *.so) to use this limited code space for executing bigger program.

- The space from 0x1000 0000 to 0x7FFF FFFF (~1.9997GB) is used to store data with certain convention for storage and structure.

- The space from 0x8000 0000 to 0xFFFF FFFF (2GB) is used for OS kernel software, kernel data and MMIO (memory mapped IO).



MARS Memory Model

MARS Memory Usage

0xffffffff	memory map limit address
0xffffffff	kernel space high address
0xffff0000	MMIO base address
0xfffeffff	kernel data segment limit address
0x90000000	.kdata base address
0x8ffffffc	kernel text limit address
0x80000180	exception handler address
0x80000000	kernel space base address
0x80000000	.ktext base address
0x7fffffff	user space high address
0x7fffffff	data segment limit address
0x7ffffffc	stack base address
0x7fffeffc	stack pointer $sp
0x10040000	stack limit address
0x10040000	heap base address
0x10010000	.data base address
0x10008000	global pointer $gp
0x10000000	data segment base address
0x10000000	.extern base address
0x0ffffffc	text limit address
0x00400000	.text base address

Kernel + MMIO

MARS $sp is not same as MIPS $sp

MARS local data location. 0x1000 0000 - 0x1000 FFFF (64K) is reserved for global variables.

MIPS Memory Usage

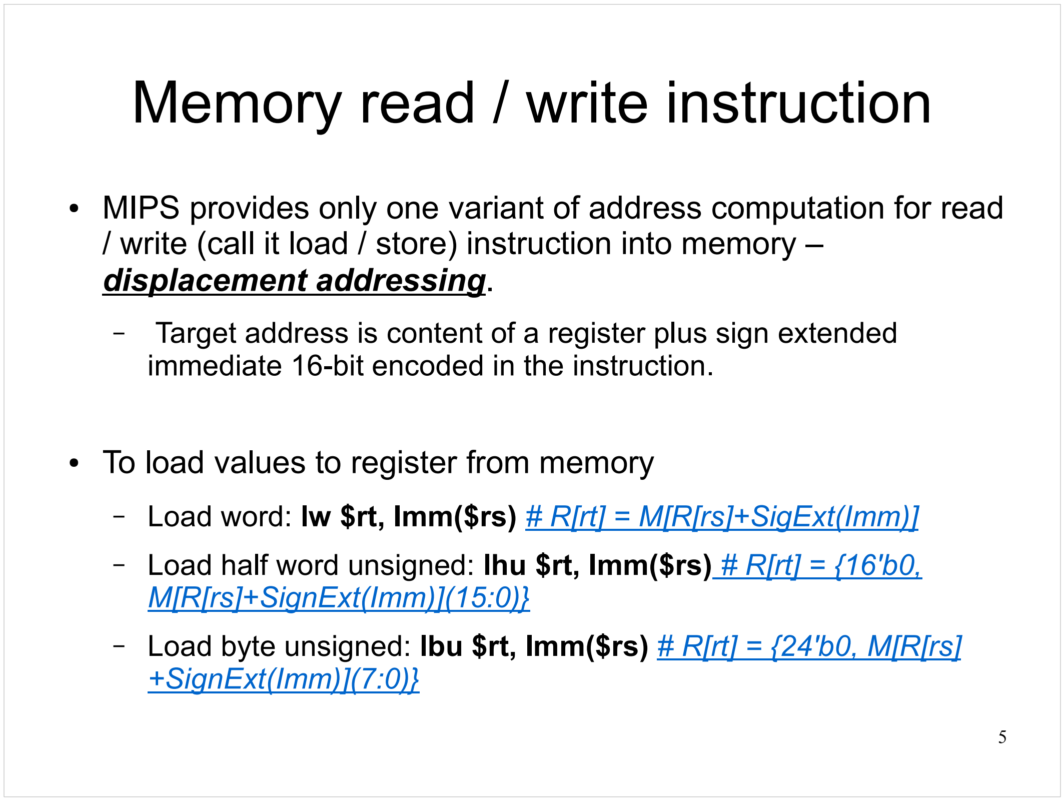
- Bring MARS memory usage map by Settings::Memory Configuration...

- MARS memory usage map is little bit different to the MIPS proposal for memory usage. The kernel and MMIO space is similar to MIPS – anything above 0x7fff ffff is kernel / mmio space. User program should be allowed max till 0x7fff ffff.

- MARS stack pointer starts at 0x7fff effc unlike MIPS power up stack pointer which starts at 0x7fff fffc. The question why should it start at 0x7fff fffc (or that matter 0x7fff effc) not at 0x7fff ffff (or 0x7fffefff) will be discussed later. Stack starts at top and grows downwards with 'push'.

- MIPS and MARS both limit user program machine code to be within 0x0040 0000 to 0x0fff ffff location. Anything beyond 0x0fff ffff is DATA to MIPS based system.

- MARS global pointer ($gp) starts at the same location that of MIPS $gp. However, all data storage after default '.data' starts at 0x1001 0000 location. MARS treats them as 'local' static variable. MARS reserves space from 0x1000 0000 to 0x1000 ffff as global data space and is defined using .extern directive.



Memory read / write instruction

- MIPS provides only one variant of address computation for read / write (call it load / store) instruction into memory – ***displacement addressing***.
  - Target address is content of a register plus sign extended immediate 16-bit encoded in the instruction.

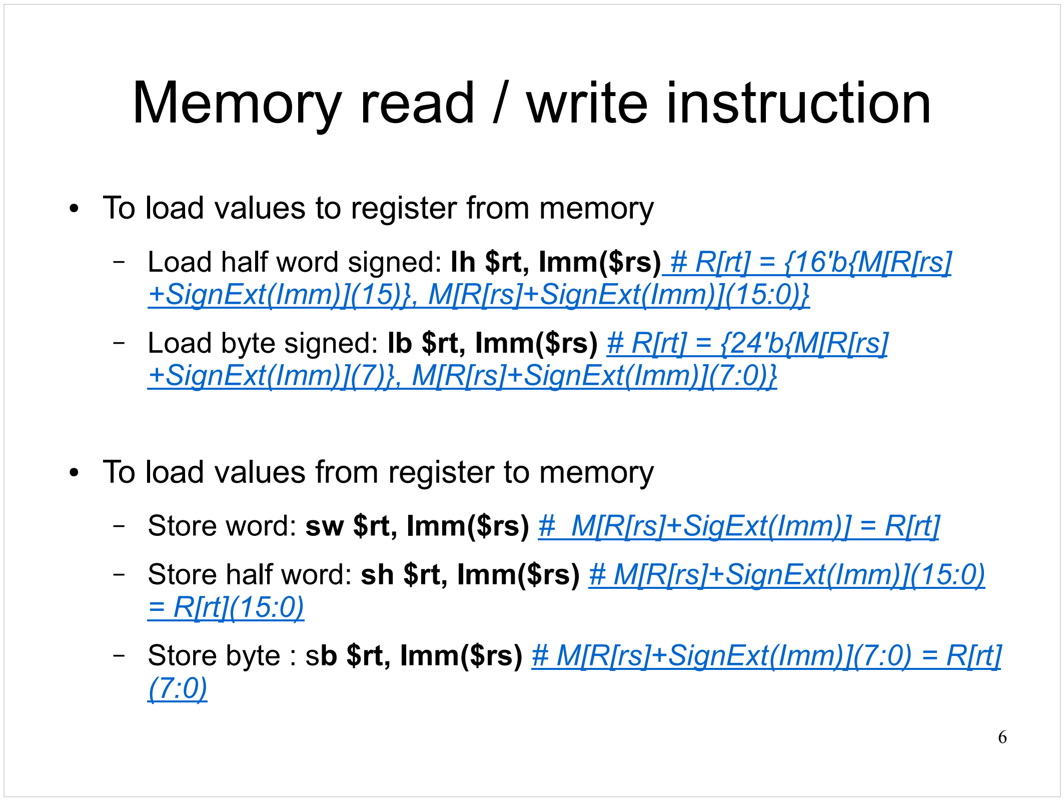
- To load values to register from memory
  - Load word: **lw $rt, Imm($rs)** *# R[rt] = M[R[rs]+SigExt(Imm)]*
  - Load half word unsigned: **lhu $rt, Imm($rs)** *# R[rt] = {16'b0, M[R[rs]+SignExt(Imm)](15:0)}*
  - Load byte unsigned: **lbu $rt, Imm($rs)** *# R[rt] = {24'b0, M[R[rs]+SignExt(Imm)](7:0)}*

5

- Also note that MARS system starts dynamic address allocation (or heap) from 0x1004 0000 and it goes upwards towards stack pointer. MARS system also limits stack growth beyond 0x1004 000 so that it can not override static data.

- The term 16'b0 means 16 bits of 0s or a half word of value 0. Similarly 24'b0 means 24 bits of 0s. Therefore an expression {16'b0, 0x1234} means 0x00001234 (16 zeros added in front of lower half word of 0x1234).

- The expression M[R[rs] + SignExt(Imm)] means the content of the memory location (32-bit wide) at address determined by adding up value of 'rs' register (the base address) with sign extended immediate value. Since the immediate value is sign extended, memory location can be lower than the base address (in case -ve value is passed as immediate) or upper than the base address (in case +ve value is passed as immediate).



Memory read / write instruction

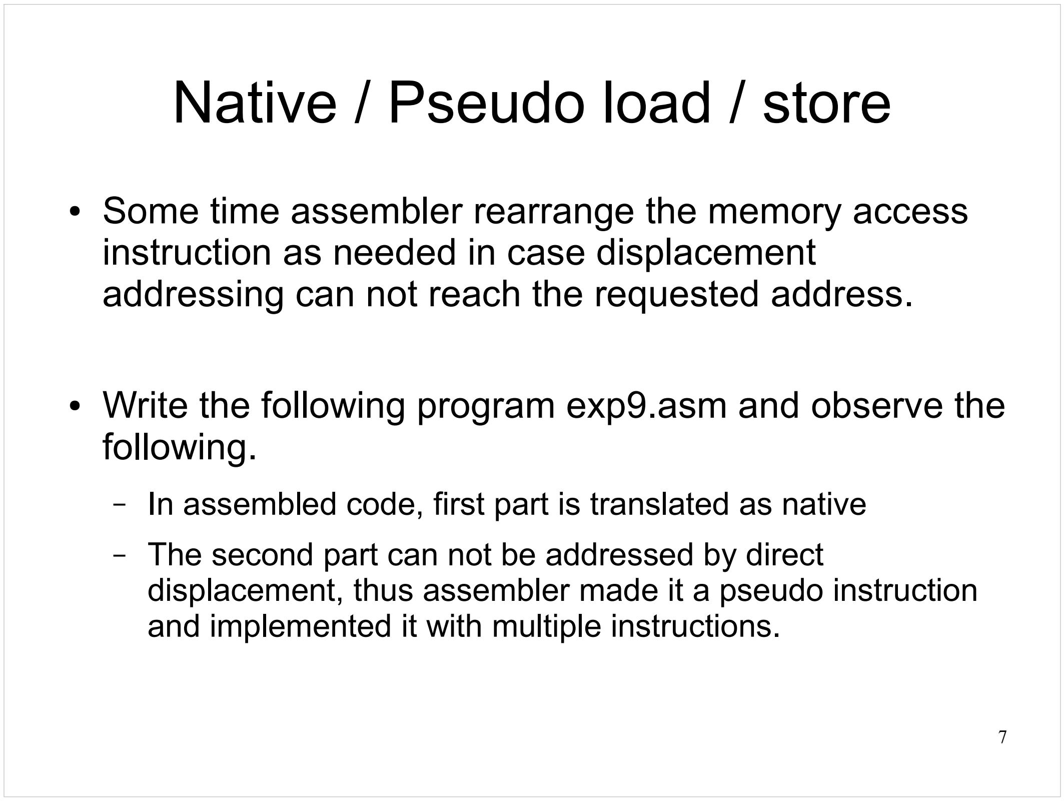
- To load values to register from memory
  - Load half word signed: **lh $rt, Imm($rs)** # R[rt] = {16'b{M[R[rs]+SignExt(Imm)](15)}, M[R[rs]+SignExt(Imm)](15:0)}
  - Load byte signed: **lb $rt, Imm($rs)** # R[rt] = {24'b{M[R[rs]+SignExt(Imm)](7)}, M[R[rs]+SignExt(Imm)](7:0)}

- To load values from register to memory
  - Store word: **sw $rt, Imm($rs)** # M[R[rs]+SigExt(Imm)] = R[rt]
  - Store half word: **sh $rt, Imm($rs)** # M[R[rs]+SignExt(Imm)](15:0) = R[rt](15:0)
  - Store byte : s**b $rt, Imm($rs)** # M[R[rs]+SignExt(Imm)](7:0) = R[rt](7:0)

6

- Note that load instruction for lower order numbers (half word, 16-bit or byte, 8-bit) has two variants – load unsigned or load signed. Since loading of register need to determine value of all the 32-bits, for loading half word or byte we need to decide what to do with rest of upper 16 (for loading half word) or 24 bits (for loading byte). One way is to load unsigned – means pad all 0s at upper bit position. The other way is to do sign extension which is replicate the MSB of the data read (16th or 8th bit for half word or byte load) into rest of the upper bit position.

- For store operation it does not make any sense of such extension of sign etc. The main reason is that memory is byte addressable – means lowest data entity can be a byte long. Therefore there is no need for extending the byte or half word value. Hence the store instruction does not have corresponding 'unsigned' revision of instruction (like sbu or shu etc.)



Native / Pseudo load / store

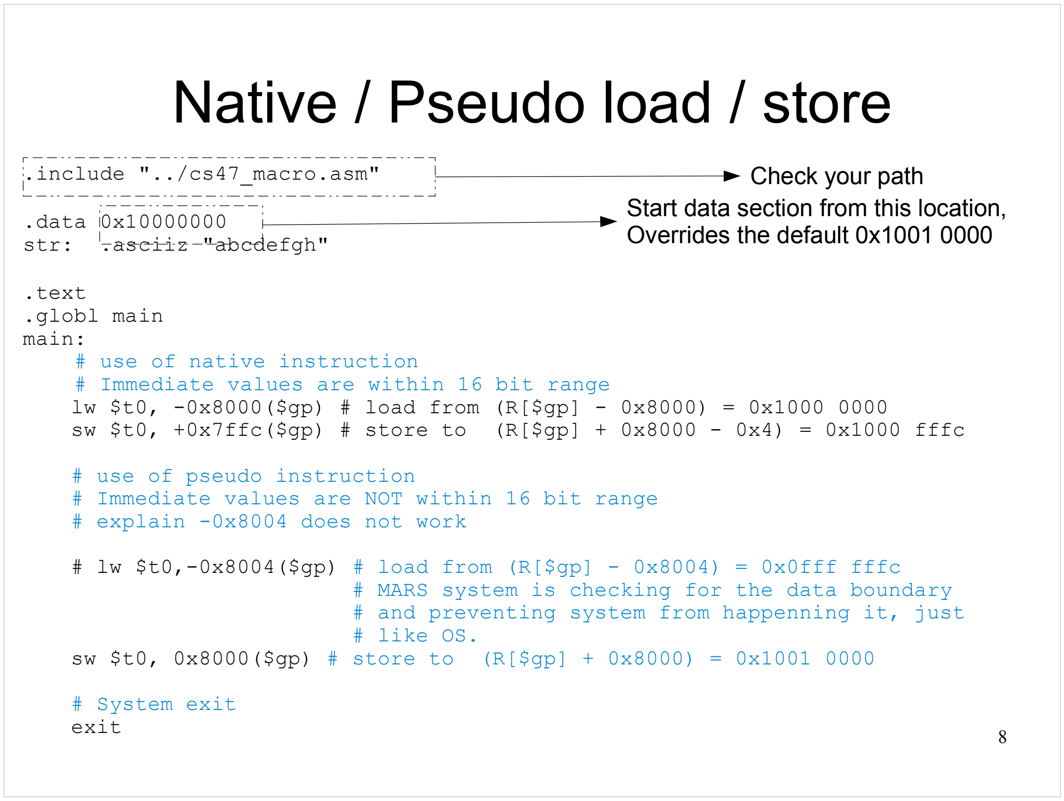
- Some time assembler rearrange the memory access instruction as needed in case displacement addressing can not reach the requested address.

- Write the following program exp9.asm and observe the following.
  - In assembled code, first part is translated as native
  - The second part can not be addressed by direct displacement, thus assembler made it a pseudo instruction and implemented it with multiple instructions.

7

- The native load store instruction supports only 16-bit signed displacements (i.e. -32KB to +32KB) from base address. Any immediate number beyond this range will need assembler intervention and translate it to multi-instruction implementation involving register $at. This is the reason that $at is assembler reserved register and user program should not use it for temporary storage.

- This experiment also shows why MARS reserves 0x1000 0000 to 0x1000 ffff address space for 'extern' (true global used by multiple program modules) variables. Anything at default '.data' location (i.e. 0x1001 0000 and above) is a program's 'local' variable. The global pointer ($gp) is set to 0x1000 8000. With 16 bit displacement using this register $gp we can address 0x1000 0000 to 0x1000 ffff. This is the reason MARS starts its local variable locations (.data default) at 0x1001 0000.



Native / Pseudo load / store

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

.data 0x10000000
str:  .asciiz "abcdefgh"

.text
.globl main
main:
    # use of native instruction
    # Immediate values are within 16 bit range
    lw $t0, -0x8000($gp) # load from (R[$gp] - 0x8000) = 0x1000 0000
    sw $t0, +0x7ffc($gp) # store to  (R[$gp] + 0x8000 - 0x4) = 0x1000 fffc

    # use of pseudo instruction
    # Immediate values are NOT within 16 bit range
    # explain -0x8004 does not work

    # lw $t0,-0x8004($gp) # load from (R[$gp] - 0x8004) = 0x0fff fffc
                          # MARS system is checking for the data boundary
                          # and preventing system from happening it, just
                          # like OS.
    sw $t0, 0x8000($gp) # store to  (R[$gp] + 0x8000) = 0x1001 0000

    # System exit
    exit
```

Check your path

Start data section from this location,
Overrides the default 0x1001 0000

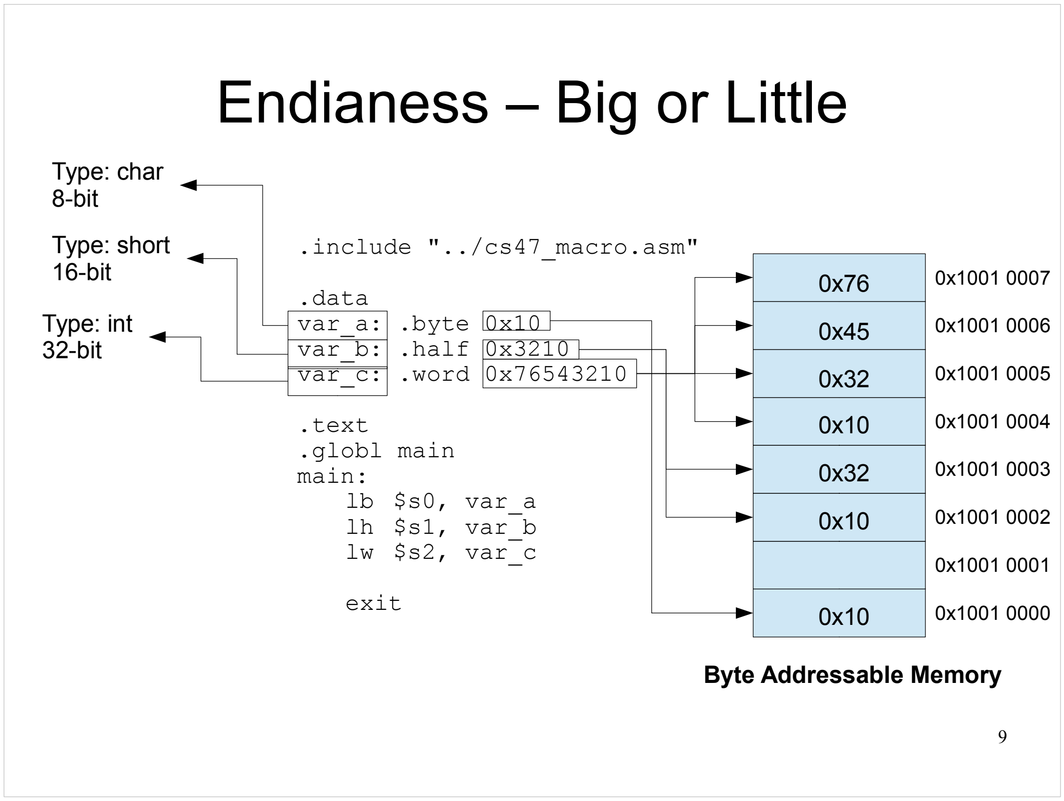
8

- The .data <start address> directive tells assembler to place the following data layout from the given start address, not from default 0x1001 0000.

- The first set of load / store has displacement values that can be represented in 16-bit. Therefore assembler can translated it directly into native machine code. However, the later set of load /store has displacement beyond 16-bit range – therefore assembler needs to treat this as pseudo instruction and combine multiple fundamental instructions to functionally replace this instruction.

- While running, the second 'lw' statement will generate run time exception. This is because the target address becomes less than 0x1000 0000, which is 'text' part from system perspective. Raw system call on a real chip without any OS would permit that. In this case, MARS is acting as OS and preventing user program to write in that area.

- Please note the replacement instructions in assembled machine code.



Endianess – Big or Little

Type: char
8-bit

Type: short
16-bit

Type: int
32-bit

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

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

.text
.globl main
main:
   lb  $s0, var_a
   lh  $s1, var_b
   lw  $s2, var_c

   exit
```

0x76	0x1001 0007
0x45	0x1001 0006
0x32	0x1001 0005
0x10	0x1001 0004
0x32	0x1001 0003
0x10	0x1001 0002
	0x1001 0001
0x10	0x1001 0000

Byte Addressable Memory

9

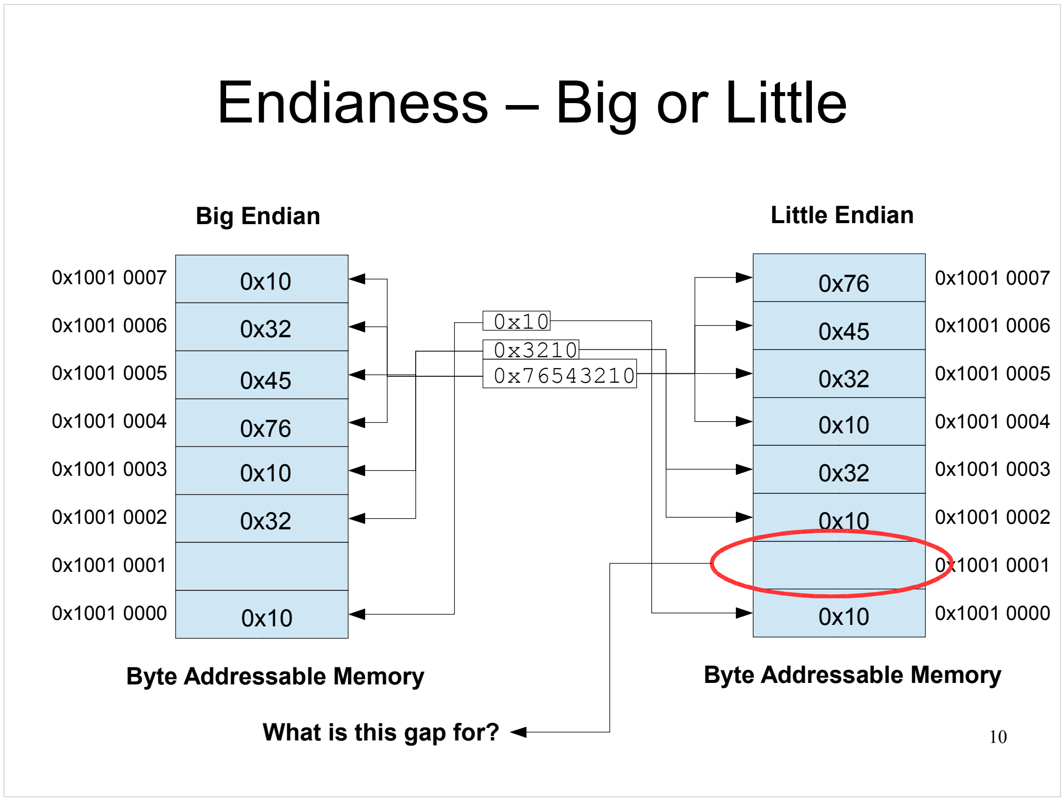
- A byte addressable memory system means individual bytes can be addressed in that memory. In other words, each address points to a unique byte storage in the memory model assumed by processor system.

- In this program three variable space is defined with three different sizes – byte (char type - 8bit), half word (short type – 16 bit) and word (int type – 32 bit).

- Once assembled, observe the memory layout by assembler in the data segment. Data bigger than byte is broken into individual bytes and stored in each address location. The var_c (which is an word is broken into 4 parts and stored in four contiguous location).

- Execute this program and observe the values at $s0, $s1 and $s2 at the end of the run. It should have values from the corresponding memory reading location.

- Also note that the 'lb <reg>, <label>' is the pseudo instruction since the native processor only supports register displacement method (lv <reg>, <displacement>(<reg>); e.g. lb $s1, 0x0($gp))



Endianess – Big or Little

Big Endian

0x1001 0007	0x10
0x1001 0006	0x32
0x1001 0005	0x45
0x1001 0004	0x76
0x1001 0003	0x10
0x1001 0002	0x32
0x1001 0001	
0x1001 0000	0x10

Byte Addressable Memory

Little Endian

0x76	0x1001 0007
0x45	0x1001 0006
0x32	0x1001 0005
0x10	0x1001 0004
0x32	0x1001 0003
0x10	0x1001 0002
	0x1001 0001
0x10	0x1001 0000

Byte Addressable Memory

What is this gap for?

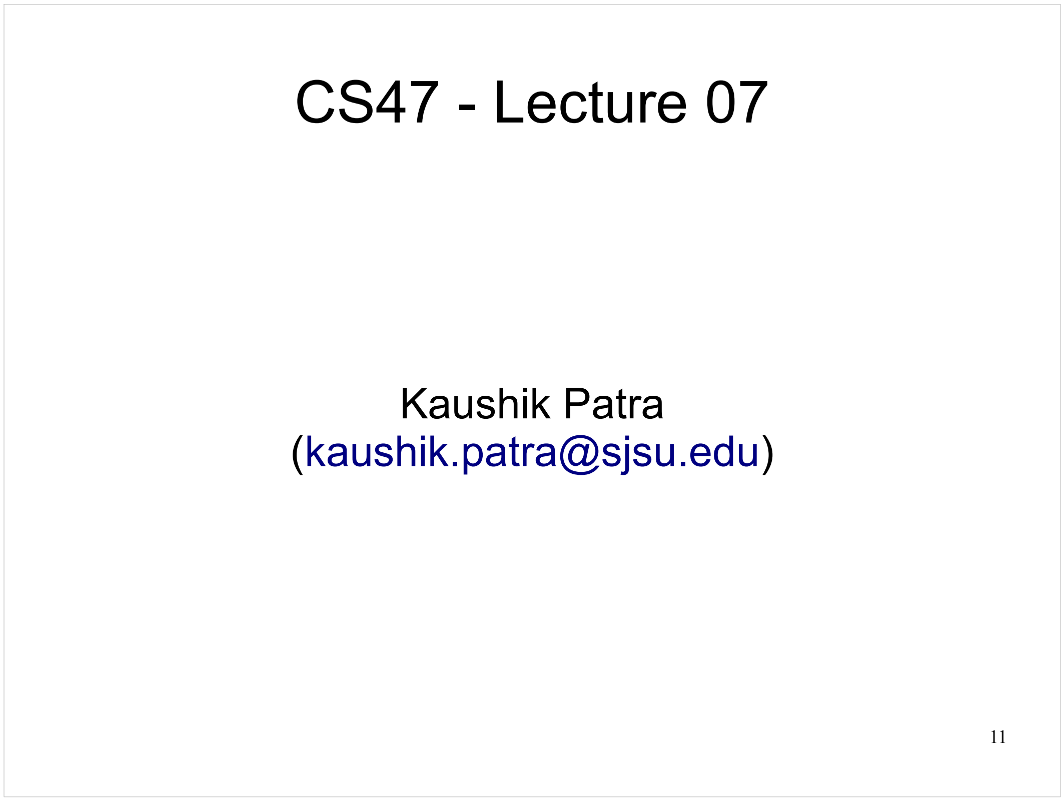
10

- Depending on the type of machine that the MARS simulation is running, multi byte data (half words or words) can have different layout for its individual bytes in the memory – one is called little endian layout and other is called big endian layout.

- In little endian type of layout, lower bytes are placed in lower address and higher bytes are placed in higher address location.

- In big endian type of layout, higher bytes are placed in lower address and lower bytes are placed in higher address location.

- Both the conventions are right in representing multi byte data. Different system designers just belong to different school of thoughts. The MIPS follows Big endian layout, where as x86 follows little endian. ARM has a very interesting idea of hardware controlled endianness. Depending on a chip external setting of a pin (0 or 1) it assumes either big or little endianness. This is to please designer from both school of thoughts.



CS47 - Lecture 07

Kaushik Patra
(kaushik.patra@sjsu.edu)

11

- Topics
  - Memory Model