CMSC 216
Introduction to Computer Systems
Assembly MIPS 1

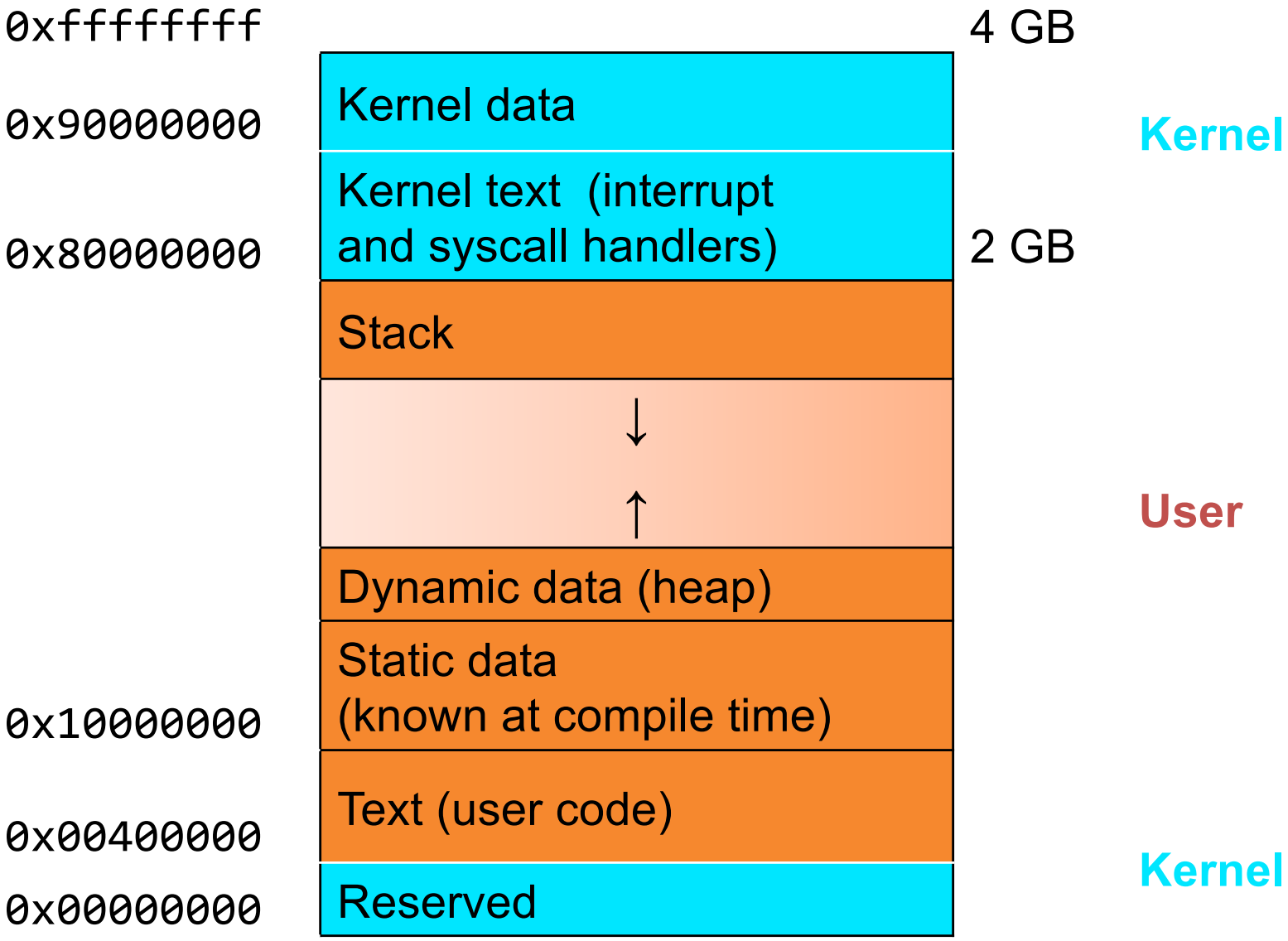
MIPS and SPIM

- We will cover the MIPS cpu
- We will run assembly programs on the SPIM simulator
- SPIM simulates a simple hardware setup
  - one MIPS cpu
  - memory
  - console: keyboard + screen
- SPIM has a simple kernel
  - provides system calls (print, read, exit) to user programs
  - handles interrupts (io) and exceptions (invalid instruction execution)
  - does not provide processes, virtual memory, filesystem, ...
  - grace: **`spim -file program.s`**
- QtSpim: SPIM with a GUI
  - use for debugging, single-step, breakpoints
  - **install locally**
- Resource: http://pages.cs.wisc.edu/~larus/HP_AppA.pdf

MIPS cpu

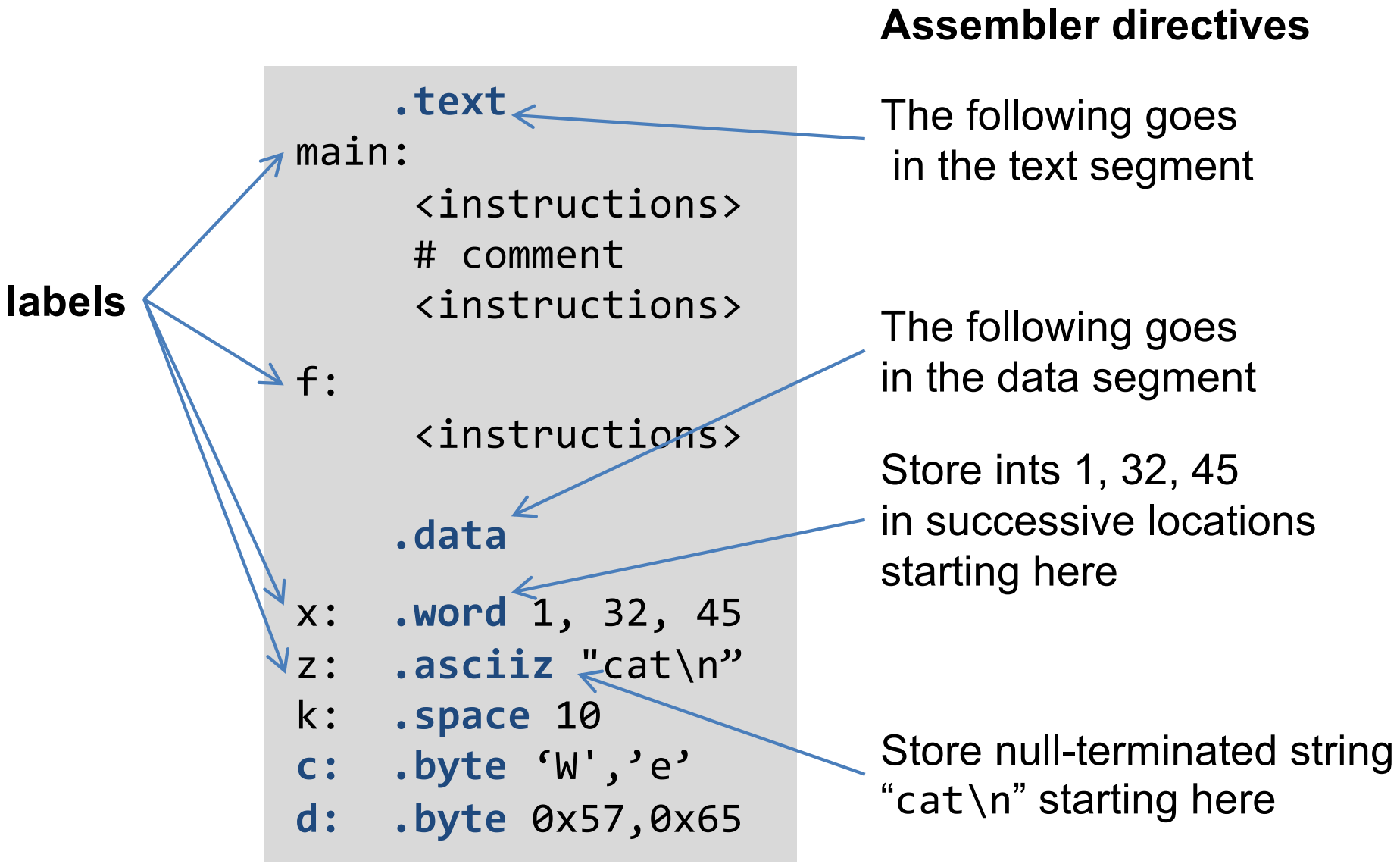
- 32-bit machine
- **word** is 32 bits (4 bytes); in memory, aligned at 4-byte boundary
- Registers are 32 bits // pc, ..., R0 – R31
- Addresses are 32 bits: can address 2^{32} bytes, or 4 GB
- Integers are 32 bits
- Each machine instruction is 32 bits
- Load/store architecture
  - Only load and store instructions access memory
  - All other instructions access registers only
- Assembly language
  - names for registers indicating usage // $v0, $v1, $a0-$a3, . . .
  - pseudo-instructions // macros

MIPS: Memory layout



www.it.uu.se/education/course/homepage/os/vt18/module-0/mips-and-mars/mips-memory-layout/

MIPS: Assembly program organization



MIPS: Some assembler directives

Directive	Meaning
`.text`	Store the following in the text segment
`.data`	Store the following in the data segment
`.word W1, ..., Wn`	Store 32-bit numbers `W1, ..., Wn` in successive memory locations
`.asciiz "str"`	Store string *`str`* and null-terminate it.

See HP_AppA.pdf for more assembler directives

MIPS: Data transfer instructions

rd, **rs**, **rb** are registers

- **lw rd, offset (rb)** // rd ← mem[offset + rb] (load word)
- **sw rs, offset (rb)** // mem[offset + rb] ← rs (store word)
- **la rd, offset (rb)** // rd ← offset + rb (load address, pseudo-instruction)
- **la rd, label** // rd ← label (load address)
- **li rd, imm_operand** // rd ← imm_operand (load immediate, pseudo-instruction)
- **move rd, rs** // rd ← rs (move)

See HP_AppA.pdf for more instructions

MIPS: arithmetic-logic instructions

- **add rd, rx, ry** // rd ← rx+ ry (addition)
  - rd: destination register
  - rx, ry: source registers
  - rd, rx, ry can be the same register
- **addu rd, rx, ry** (addition unsigned)
- **sub rd, rx, ry** // rd ← rx - ry (subtraction)
- **mul " " "** // rd ← rx * ry (multiply, 32-bit result)
- **and " " "** // rd ← rx & ry (bitwise and)
- **or " " "** // rd ← rx | ry (bitwise or)
- **sll " " "** // rd ← rx << ry (shift left by ry),
- **srl " " "** // rd ← rx >> ry (shift right by ry)
- Third operand can be **immediate** value (constant or literal)
  - **add rd, rx, 100** // rd ← rx + 100

MIPS: Unconditional jump instructions

- **j label** // jump to **label** ; $pc ← label

- **jr r** // jump to address in register **r** ; $pc ← r
  // used for switch, function return

- **jal label** // jump and link to **label**
  // save the address of the next instruction in **$ra**
  // jump to **label**
  // used for function call

MIPS: Conditional branch instructions

- Two-register condition
  - **beq r1, r2, label** // branch to **label** if **r1 == r2**
  - **bne r1, r2, label** // branch to **label** if **r1 != r2**
  - **bge r1, r2, label** // branch to **label** if **r1 >= r2**
  - **bgt r1, r2, label** // branch to **label** if **r1 > r2**
  - **ble r1, r2, label** // branch to **label** if **r1 <= r2**
  - **. . .**
- Second register operand can be a literal
  - **beq r1, 100, label**
- One-register condition
  - **beqz r1, label** // branch to **label** if **r1 == 0**
  - **. . .**

MIPS: syscall and exception handling

- **syscall instruction**
  - "jump" to address **0x80000180** (in kernel text)
  - used as the final step in requesting kernel service (eg, print, read)
- The kernel code at that address reads registers to determine the requested service, handles the request, and returns to the "caller".
- **System call**

```
$v0 ← integer code
$a0-$a3 ← arguments (if any)
syscall
```

- The above effect also happens when
  - cpu does an invalid instruction execution
  - an io-device sends an **interrupt** to the cpu (via bus)

System calls (in SPIM)

Service	code in $v0	arguments	result in $v0
print_int	1	$a0: int to print	none
print_string	4	$a0: address of null-terminated string to print	none
print_char	11	$a0: char to print	none
read_int	5	none	$v0 = int read
read_char	12	none	$v0 = char read
read_string	8	$a0 / $a1: address /length of string input buffer	none
exit	10	none	none

See HP_AppA.pdf for more system calls