4. Instruction Set Architecture – from C to assembly – Part 1

EECS 370 – Introduction to Computer Organization – Winter 2009

Prof. Todd Austin & Prof. Marios Papaefthymiou

EECS Department
University of Michigan, Ann Arbor, USA

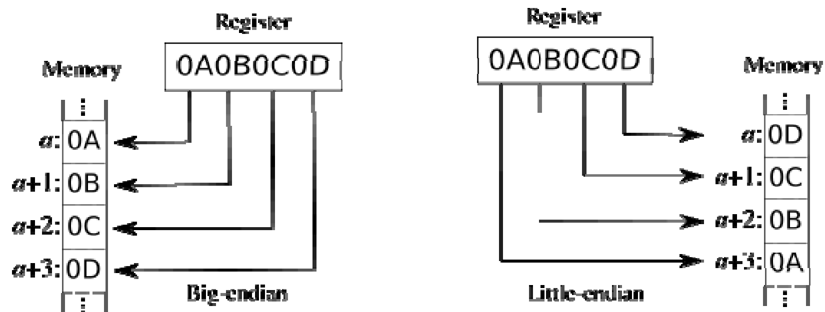
© T. Austin & M. Papaefthymiou, 2009
The material in this presentation cannot be copied in any form without our written permission

Recap: LC-2K and MIPS ISAs

- LC2K
  - 32-bit word-addressable, 65,536 words of memory
  - 8 registers
  - 8 instructions: add, nand, lw, sw, beq, jalr, noop, halt
  - Instruction formats: R-type, I-type (16-bit immediate)
- MIPS
  - 32-bit byte-addressable
  - Lots of arithmetic operations
  - Memory addressing: Base+displacement (16-bit immediate)
  - Memory operations on bytes, halfwords (2 bytes), and words (4 bytes)
  - Big-Endian

Addressing: Big Endian vs Little Endian

- Endianness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - MIPS is Big Endian



© T. Austin & M. Papaefthymiou – 2009

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3: LC-2K and MIPS architecture
- **Lecture 4 and 5: Converting C to assembly:**
  - **Data structures**
  - **Branch instructions**
  - **Calling functions / passing arguments**
  - **Caller / callee save registers**
- Lecture 6: Translation software; libraries, VMs

© T. Austin & M. Papaefthymiou – 2009

Converting C to Assembly

- Memory layout → memory addresses
- Branches
- Procedure calls
- Expression trees
- Register allocation

© T. Austin & M. Papaefthymiou – 2009

Converting C to assembly – example 1

DATA LAYOUT

Write MIPS Assembly Code for the following C Expression:

C: a = b + names[i];

Assume that **a** is in $1, **b** is in $2, **i** is in $3, and the array **names** starts at address 1000 and holds 32-bit integers.

```
multi   $5, $3, 4        // calculate array offset
lw      $4,  1000 ($5)   // load names[i]
add     $1,  $2,  $4     // calculate b + names[i]
```


© T. Austin & M. Papaefthymiou – 2009

Converting C to assembly – example 2

DATA LAYOUT

Write MIPS Assembly Code for the following C Expression:

class { int a; char b, c; } y;

y.a = y.b + y.c;

Assume that a pointer to y is in $1.

```
lb    $2, 4 ($1)   // load y.b
lb    $3, 5 ($1)   // load y.c
add   $4, $2, $3   // calculate y.b+y.c
sw    $4, 0 ($1)   // store y.a
```


© T. Austin & M. Papaefthymiou – 2009

Calculating Load/Store Addresses for Variables

DATA LAYOUT

```
short a[100];
char b;
int c;
double d;
short e;
struct {
    char f;
    int g[1];
    char h;
} i;
```

Problem: Assume data memory starts at address 100, calculate the total amount of memory needed.

a = 2 bytes * 100 = 200
b = 1 byte
c = 4 bytes
d = 8 bytes
e = 2 bytes
i = 1 + 4 + 1 = 6 bytes

total = 221, right or wrong?

© T. Austin & M. Papaefthymiou – 2009

Memory layout of variables

DATA LAYOUT

- Cannot arbitrarily pack variables into memory → Need to worry about alignment
- "Golden" rule – Address of a variable is aligned based on the size of the variable
  - **char** is byte aligned (any addr is fine)
  - **short** is half-word aligned (LSB of addr must be 0)
  - **int** is word aligned (2 LSBs of addr must be 0)

*Struct*ure alignment

DATA LAYOUT

- Each field is laid out in the order it is declared using the Golden Rule for aligning
- Identify largest field
  - Starting address of overall struct is aligned based on the largest field
  - Size of overall struct is a multiple of the largest field
  - Reason for this is so can have an array of structs

DATA LAYOUT

Structure Example

```
struct {
        char w;
        int x[3]
        char y;
        short z;
}
```

The largest field is **int** (4 bytes), hence:

- struct size is multiple of 4
- struct's starting addr is word aligned

Assume struct starts at location 1000,

char w → 1000
x[0] → 1004-1007, x[1] → 1008 – 1011, x[2] → 1012 – 1015
char y → 1016
short z → 1018 – 1019

Total size = 20 bytes!

DATA LAYOUT

Earlier Example – 2nd Try

Assume data memory starts at address 100

Declaration	Layout
short a[100];	→ 200 bytes → 100-299
char b;	→ 1 byte → 300-300
int c;	→ 4 bytes → 304-307
double d;	→ 8 bytes → 312-319
short e;	→ 2 bytes → 320-321
struct {	→ largest field is 4 bytes → start at 324
char f;	→ 1 byte → 324-324
int g[1];	→ 4 bytes → 328 - 331
char h;	→ 1 byte → 332-332
} i;	→ struct size is 12 bytes, spanning 324 – 335

236 bytes total!!

Class Problem 1

DATA LAYOUT

- How much memory is required for the following data, assuming that the data starts at address 200?

```
int a;
struct {double b, char c, int d} e;
char *f;
short g[20];
```


© T. Austin & M. Papaefthymiou – 2009

MIPS Sequencing Instructions

BRANCH

- Sequencing instructions change the flow of instructions that are executed.
  - This is achieved by modifying the program counter.
- Conditional branches
  - If (condition_test) goto target_address
    - *condition_test* compares two operands (registers)
    - *target_address* is a 16-bit displacement on current PC+4
  - **beq $x, $y, val**
    - if ($x == $y) then PC = PC + 4 + val else PC = PC + 4

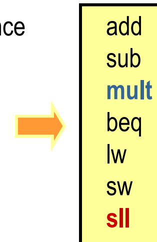
© T. Austin & M. Papaefthymiou – 2009

BRANCH

Setting the Displacement Field

- Target_address is a 16-bit displacement on current PC+4
  - Target = PC + 4 + displacement
    - beq $4, $0, 8 // branch 3 instr ahead if $4 = = 0
    - beq $4, $0, -8 // branch 1 instruction back if $4 == 0
    - beq $4, $0, -4 // Infinite loop if $4 == 0

Example code sequence



BRANCH

Other branching instructions

- bne $2, $3, offset // branch to offset+PC+4 if $2 ≠ $3
- blt $2, $3, offset // branch to offset+PC+4 if $2 < $3
- bge $2, $3, offset // pseudo-instruction $2 ≥ $3
- j target // jump to target (pseudo-direct addressing mode)
  // uses a 26-bit target address concatenated to top 6
  // bits of PC
- jr $3 // jump to address in $3 -- when is this useful?
- jal target // put PC+4 into register $31 and jump to target address

BRANCH

Branch - example

Convert the following C code into MIPS assembly (assume x is in $1, y in $2):

```
     int x, y;
     if (x == y)
          x++;
(L1) else
          y++;
(L2) …
```

Using Labels

```
      bne $1, $2, L1
      addi $1, $1, 1
      beq $0, $0, L2
L1:   addi $2, $2, 1
L2:   …
```

Without Labels

```
bne $1, $2, 8
addi $1, $1, 1
beq $0, $0, 4
addi $2, $2, 1
```

Assemblers must deal with labels and assign displacements

BRANCH

Loop - example

```
// assume all variables are integers
// i is in $1, start of a is 500, sum is in $2
for (i=0 ; i < 100 ; i++) {
     if (a[i] > 0) {
          sum += a[i];
     }
}
```

```
          add     $1, $0, $0
          addi    $4, $0, 400
Loop1:    bge     $1, $4, endLoop
          lw      $5, 500($1)
          ble     $5, $0, endIf
          add     $2, $2, $5
endIf:    addi    $1, $1, 4
          j       Loop1
endLoop:
```

Converting function calls to assembly code

FUNCTION CALLS

C: printf("hello world\n");

- Need to pass parameters to the called function (printf)
- Need to save return address of caller
- Need to save register values
- Need to jump to printf

Execute instructions for printf()
Jump to return address

- Need to get return value (if used)
- Restore register values

© T. Austin & M. Papaefthymiou – 2009

Task 1: Passing parameters

FUNCTION CALLS

- Where should you put all of the parameters?
  - Registers?
    - Fast access but few in number and wrong size for some objects
  - Memory?
    - Good general solution but where?

- MIPS answer:
  - Registers and memory.
    - Put the first few parameters in registers (if they fit) ($4 - $7)
    - Put the rest in memory on the ***call stack***

  - Example:
    addi $4, $0, 1000 // put address of char array "hello world" in $4

© T. Austin & M. Papaefthymiou – 2009

Call stack

FUNCTION CALLS

- MIPS conventions (and most other processors) allocate a region of memory for the call stack
  - This memory is used to manage all the storage requirements to simulate function call semantics
    - Parameters (that were not passed through registers)
    - Local variables
    - Temporary storage (when you run out of registers and need somewhere to save a value)
    - Return address
    - Etc.
- Sections of memory on the call stack [a.k.a. **stack frames**] are allocated when you make a function call, and de-allocated when you return from a function.

© T. Austin & M. Papaefthymiou – 2009

MIPS Memory Map

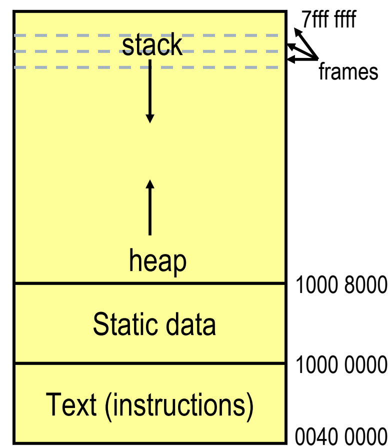
FUNCTION CALLS

Stack: starts at 0x7fff ffff and grows down to lower addresses. Bottom of the stack resides in the $sp register ($29 by convention).

Heap: starts at 0x1000 8000 and grows up to higher addresses. Allocation done explicitly with malloc(). Deallocation with free(). Runtime error if no free memory before running into $sp address.

Static: starts at 0x1000 0000. Holds all global variables and those locals explicitly declared as "static".

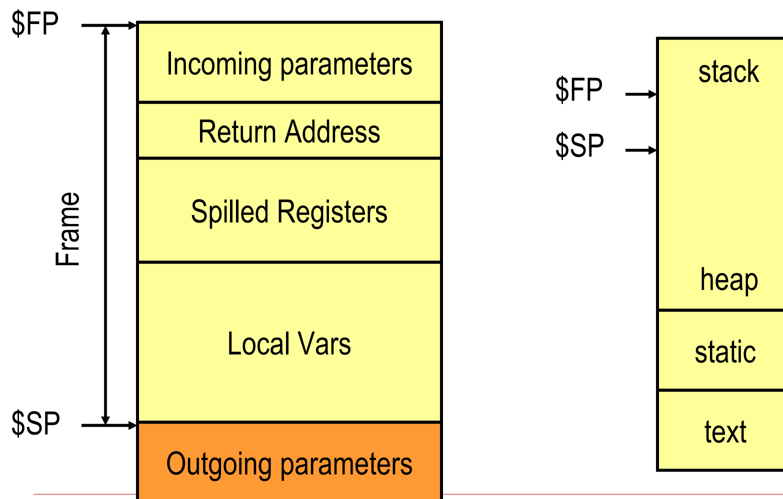
Text: starts at 0x0040 0000. Holds all instructions in the program (except for Dynamically linked library routines DLLs)



© T. Austin & M. Papaefthymiou – 2009

FUNCTION CALLS

The MIPS Stack Frame



© T. Austin & M. Papaefthymiou – 2009

FUNCTION CALLS

Allocating space to local variables

- Local variables (by default) are created when you enter a function, and disappear when you leave
  - Technical terminology: local variables are placed in the automatic storage class (as opposed to the static storage class used for globals).
- Automatics are allocated on the call stack
  - How?
    by incrementing (or decrementing) the pointer to the top of the call stack
  - sub $sp, $sp, 12 // allocate space for 3 integer locals
    add $sp, $sp, 12 // de-allocate space for locals

© T. Austin & M. Papaefthymiou – 2009

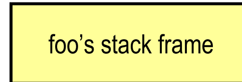
FUNCTION CALLS

The stack grows as functions are called

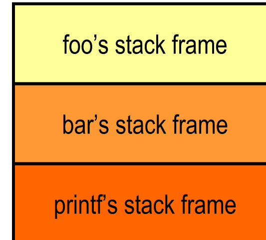
```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

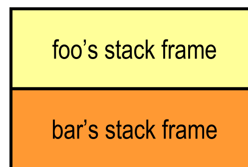
inside foo



bar calls printf



foo calls bar



The University of Michigan
© T. Austin & M. Papaefthymiou – 2009

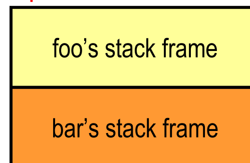
FUNCTION CALLS

The stack shrinks as functions return

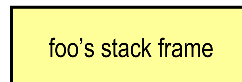
```
void foo()
{
   int x, y[2];
   bar(x);
}

void bar(int x)
{
   int a[3];
   printf();
}
```

printf returns



bar returns

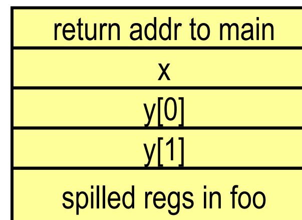


The University of Michigan
© T. Austin & M. Papaefthymiou – 2009

Stack frame contents

```
void foo()
{
    int x, y[2];
    bar(x);
}

void bar(int x)
{
    int a[3];
    printf();
}
```

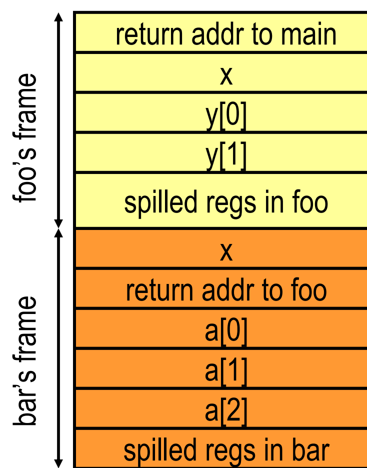


© T. Austin & M. Papaefthymiou – 2009

Stack frame contents (2)

```
void foo()
{
    int x, y[2];
    bar(x);
}

void bar(int x)
{
    int a[3];
    printf();
}
```

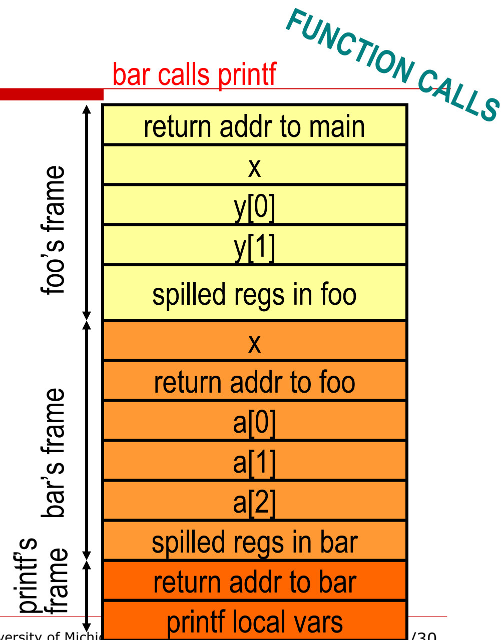


© T. Austin & M. Papaefthymiou – 2009

Stack frame contents (3)

```
void foo()
{
    int x, y[2];
    bar(x);
}

void bar(int x)
{
    int a[3];
    printf();
}
```



Assigning variables to memory spaces

Code	Comment
int w;	w goes in static, as it's a global
void foo(int x)	x goes on the stack, as it's a parameter
{	
static int y[4];	y goes in static, 1 copy of this!!
char *p;	p goes on the stack
p = malloc(10);	allocate 10 bytes on heap, ptr set to the address
…	
printf("%s\n", p);	string goes in static, pointer to string on stack, p goes on stack
}