Week 06

Exercise 1: Function to sum values in array

1/30

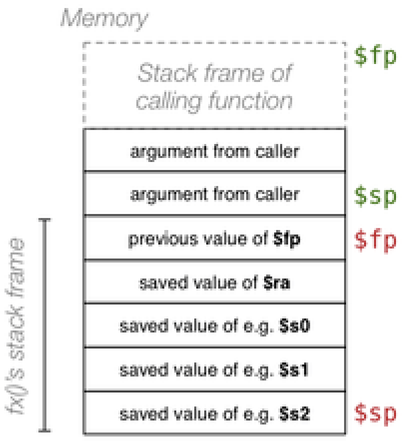
Implement a MIPS version of the following:

```
int array[10] = {5,4,7,6,8,9,1,2,3,0};

int main(void)
{
   printf("%d\n", sumOf(array,0,9));
   return 0;
}

int sumOf(int a[], int lo, int hi)
{
   if (lo > hi)
      return 0;
   else
      return a[lo] + sumOf(a,lo+1,hi);
}
```

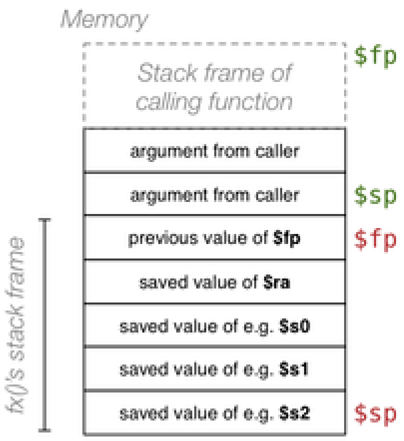
Reminder ...




```
# start of fx() function
fx:
# fx()'s prologue
sw   $fp, -4($sp)
sw   $ra, -8($sp)
sw   $s0, -12($sp)
sw   $s1, -16($sp)
sw   $s2, -20($sp)
la   $fp, -4($sp)
add  $sp, $sp, -20
# rest of fx()'s code
```

initial values: $fp,$sp
final values: $fp,$sp

Alternatively ...




```
# start of fx() function
fx:
# fx()'s prologue
addi $sp, $sp, -4
sw   $fp, ($sp)
move $fp, $sp
addi $sp, $sp, -4
sw   $ra, ($sp)
addi $sp, $sp, -4
sw   $s0, ($sp)
addi $sp, $sp, -4
sw   $s1, ($sp)
addi $sp, $sp, -4
sw   $s2, ($sp)
# rest of fx()'s code
```

initial values: $fp,$sp   final values: $fp,$sp

Data Structures and MIPS

4/30

C data structures and their MIPS representations:

- `char` ... as byte in memory, or low-order byte in register
- `int` ... as word in memory, or whole register
- `double` ... as two-words in memory, or `$f?` register
- arrays ... sequence of memory bytes/words, accessed by index
- structs ... chunk of memory, accessed by fields/offsets
- linked structures ... struct containing address of other struct

A `char`, `int` or `double`

- could be implemented in register if used in small scope
- could be implemented on stack if local to function
- could be implemented in `.data` if need longer persistence

Static vs Dynamic Allocation

Static allocation:

- uninitialised memory allocated at compile/assemble-time, e.g.

```
int  val;                val: .space 4
char str[20];            str: .space 20
int  vec[20];            vec: .space 80
```

- initialised memory allocated at compile/assemble-time, e.g.

```
int val = 5;                   val: .word 5
int arr[4] = {9,8,7,6};        arr: .word 9, 8, 7, 6
char *msg = "Hello\n";         msg: .asciiz "Hello\n"
```

... Static vs Dynamic Allocation

Dynamic allocation (i):

- variables local to a function

Prefer to put local vars in registers, but if cannot ...

- space allocated on stack during function prologue
- referenced during function relative to `$fp`
- space relcaimed from stack in function epilogue

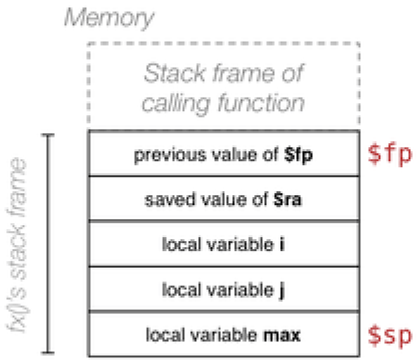
Example:

```
int fx(int a[])
{
   int i, j, max;

   i = 1;  j = 2;  max = 0;
   ...
}
```

... Static vs Dynamic Allocation

Example of local variables on the stack:




```
# start of fx() function
fx:
# fx()'s prologue
# save $fp, $ra; fix $fp, $sp
# allocate space for i, j, max
add  $sp, $sp, -12
# inside fx()'s code
li   $t0, 1        # i = 1;
sw   $t0, -8($fp)
li   $t0, 2        # j = 2;
sw   $t0, -12($fp)
sw   $0, -16($fp)  # max = 0;
…
# fx()'s epilogue
add  $sp, $sp, 12
lw   $ra, ($sp)
…
```

... Static vs Dynamic Allocation

8/30

Dynamic allocation (ii):

- uninitialised block of memory allocated at run-time

```
int  *ptr = malloc(sizeof(int));
char *str = malloc(20*sizeof(char));
int  *vec = malloc(20*sizeof(int));

*ptr = 5;
strcpy(str, "a string");
vec[0] = 1;  // or  *vec = 1;
vec[1] = 6;
```

- initialised block of memory allocated at run-time

```
int *vec = calloc(20, sizeof(int));
// vec[i] == 0, for i in 0..19
```

... Static vs Dynamic Allocation

9/30

SPIM doesn't provide `malloc()`/`free()` functions

- but provides a `syscall` to extend `.data`
- before `syscall`, set `$a0` to the number of bytes requested
- after `syscall`, `$v0` holds start address of allocated chunk

Example:

```
li   $a0, 20    # $v0 = malloc(20)
li   $v0, 9
syscall
move $s0, $v0   # $s0 = $v0
```

Cannot access allocated data by name; need to retain address.

No way to free allocated data, and no way to align data appropriately

---

Exercise 2: Implementing malloc()

10/30

Using `syscall` 9, implement a function like `malloc()`

- input parameter (`$a0`) is number of bytes to allocate
- result (`$v0`) is address of first allocated byte

Implement a function like `calloc()`

- first parameter (`$a0`) is number of items to allocate
- second parameter (`$a1`) is size (bytes) of each item
- result (`$v0`) is address of first allocated item

1-d Arrays in MIPS

11/30

Can be named/initialised as noted above:

```
vec:  .space 40
# could be either int vec[10] or char vec[40]

nums: .word 1, 3, 5, 7, 9
# int nums[6] = {1,2,3,5,7,9}
```

Can access elements via index or cursor (pointer)

- either approach needs to account for size of elements

Arrays passed to functions via pointer to first element

- must also pass array size, since not available elsewhere

See `sumOf()` exercise for an example of passing an array to a function

---

... 1-d Arrays in MIPS

12/30

Scanning across an array of `N` elements using index

```
# int vec[10] = {...};
# int i;
# for (i = 0; i < 10; i++)
#    printf("%d\n", vec[i]);

   li   $s0, 0              # i = 0
   li   $s1, 10
loop:
   bge  $s0, $s1, end_loop  # if (i >= 10) break
   li   $t0, 4
   mul  $t0, $s0, $t0       # index -> byte offset
   lw   $a0, vec($t0)       # a0 = vec[i]
   jal  print               # print a0
   addi $s0, $s0, 1         # i++
   j    loop
end_loop:
```

Assumes the existence of a `print()` function to do `printf("%d\n",x)`

... 1-d Arrays in MIPS

13/30

Scanning across an array of `N` elements using cursor

```
# int vec[10] = {...};
# int *cur, *end = &vec[10];
# for (cur = vec; cur < end; cur++)
#    printf("%d\n", *cur);

   la   $s0, vec            # cur = &vec[0]
   la   $s1, vec+40         # end = &vec[10]
loop:
   bge  $s0, $s1, end_loop  # if (cur >= end) break
   lw   $a0, ($s0)          # a0 = *cur
   jal  print               # print a0
   addi $s0, $s0, 4         # cur++
```

```
    j     loop
end_loop:
```

Assumes the existence of a `print()` function to do `printf("%d\n",x)`

... 1-d Arrays in MIPS

14/30

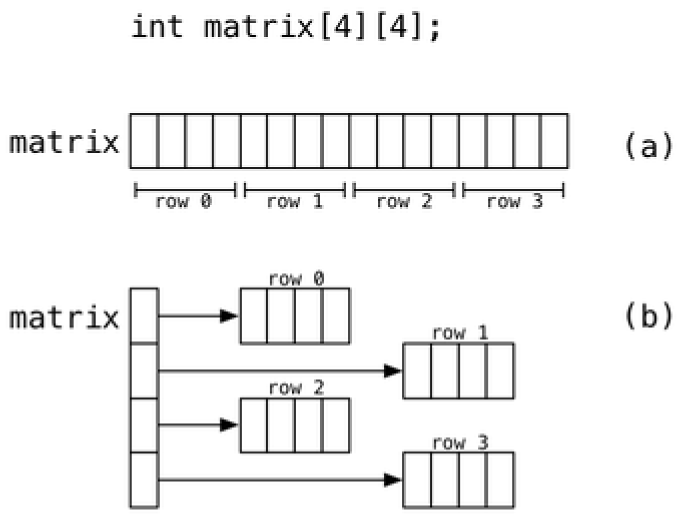
Arrays that are local to functions are allocated space on the stack

```
fun:                              int fun(int x)
   # prologue                     {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)                   // push a[] onto stack
   addi $sp, $sp, -40                int a[10];
   move $s0, $sp                     int *s0 = a;
   # function body
   ... compute ...                   // compute using s0
   # epilogue                        // to access a[]
   addi $sp, $sp, 40                 // pop a[] off stack
   lw   $ra, ($sp)
   addi $sp, $sp, 4
   lw   $fp, ($sp)
   addi $sp, $sp, 4
   jr   $ra                       }
```

2-d Arrays in MIPS

15/30

2-d arrays could be represented two ways:



... 2-d Arrays in MIPS

16/30

Representations of `int matrix[4][4]` ...

```
# for strategy (a)
matrix: .space 64
# for strategy (b)
row0:    .space 16
row1:    .space 16
row2:    .space 16
row3:    .space 16
matrix: .word row0, row1, row2, row3
```

Now consider summing all elements

```
int i, j, sum = 0;
for (i = 0; i < 4; i++)
```

```
    for (j = 0; j < 4; j++)
       sum += matrix[i][j];
```

... 2-d Arrays in MIPS

17/30

Computing sum of all elements for strategy (a) `int matrix[4][4]`

```
   li  $s0, 0             # sum = 0
   li  $s1, 4             # s1 = 4
   li  $s2, 0             # i = 0
loop1:
   beq  $s2, $s1, end1    # if (i >= 4) break
   li   $s3, 0            # j = 0
loop2:
   beq  $s3, $s1, end2    # if (j >= 4) break
   mul  $t0, $s2, 16      # off = 4*4*i + 4*j
   mul  $t1, $s3, 4       #  matrix[i][j] is
   add  $t0, $t0, $t1     #  done as *(matrix+off)
   lw   $t0, matrix($t0)  # t0 = matrix[i][j]
   add  $s0, $s0, $t0     # sum += t0
   addi $s3, $s3, 1       # j++
   j    loop2
end2:
   addi $s2, $s2, 1       # i++
   j    loop1
end1:
```

... 2-d Arrays in MIPS

18/30

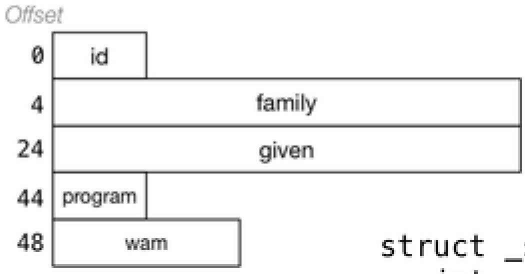
Computing sum of all elements for strategy (b) `int matrix[4][4]`

```
   li  $s0, 0             # sum = 0
   li  $s1, 4             # s1 = 4 (sizeof(int))
   li  $s2, 0             # i = 0
loop1:
   beq  $s2, $s1, end1    # if (i >= 4) break
   li   $s3, 0            # j = 0
   mul  $t0, $s2, 4       # off = 4*i
   lw   $s4, matrix($t0)  # row = &matrix[i][0]
loop2:
   beq  $s3, $s1, end2    # if (j >= 4) break
   mul  $t0, $s3, 4       # off = 4*j
   add  $t0, $t0, $s4     # int *p = &row[j]
   lw   $t0, ($t0)        # t0 = *p
   add  $s0, $s0, $t0     # sum += t0
   addi $s3, $s3, 1       # j++
   j    loop2
end2:
   addi $s2, $s2, 1       # i++
   j    loop1
end1:
```

Structs in MIPS

19/30

C `structs` hold a collection of values accessed by name




```
struct _student {
   int    id;
   char   family[20];
   char   given[20];
   int    program;
   double wam;
};
```

... Structs in MIPS

20/30

C `struct` definitions effectively define a new type.

```
// new type called struct _student
struct _student {...};
// new type called Student
typedef struct _student Student;
```

Instances of structures can be created by allocating space:

```
                  // sizeof(Student) == 56
stu1:             Student stu1;
   .space 56
stu2:             Student stu2;
   .space 56
stu:
   .space 4       Student *stu;
```

... Structs in MIPS

21/30

Accessing structure components is by offset, not name

```
li   $t0  5012345
sw   $t0, stu1+0       # stu1.id = 5012345;
li   $t0, 3778
sw   $t0, stu1+44      # stu1.program = 3778;
la   $s1, stu2         # stu = & stu2;
li   $t0, 3707
sw   $t0, 44($s1)      # stu->program = 3707;
li   $t0, 5034567
sw   $t0, 0($s1)       # stu->id = 5034567;
```

... Structs in MIPS

22/30

Structs that are local to functions are allocated space on the stack

```
fun:                          int fun(int x)
   # prologue                 {
   addi $sp, $sp, -4
   sw   $fp, ($sp)
   move $fp, $sp
   addi $sp, $sp, -4
   sw   $ra, ($sp)               // push onto stack
   addi $sp, $sp, -56            Student st;
   move $t0, $sp                 Student *t0 = &st;
   # function body
   ... compute ...               // compute using t0
   # epilogue                    // to access struct
```

```
    addi $sp, $sp, 56             // pop st off stack
    lw   $ra, ($sp)
    addi $sp, $sp, 4
    lw   $fp, ($sp)
    addi $sp, $sp, 4
    jr   $ra                  }
```

... Structs in MIPS

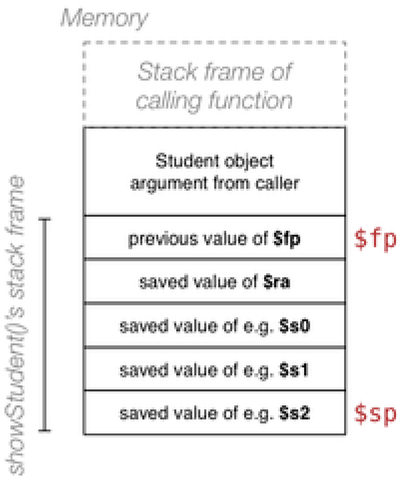
C can pass whole structures to functions, e.g.

```
# Student stu; ...
# // set values in stu struct
# showStudent(stu);

   .data
stu: .space 56
   .text
   ...
   la   $t0, stu
   addi $sp, $sp, -56    # push Student object onto stack
   lw   $t1, 0($t0)      # allocate space and copy all
   sw   $t1, 0($sp)      # values in Student object
   lw   $t1, 4($t0)      # onto stack
   sw   $t1, 4($sp)
   ...
   lw   $t1, 52($t0)     # and once whole object copied
   sw   $t1, 52($sp)
   jal  showStudent      # invoke showStudent()
   ...
```

... Structs in MIPS

Accessing `struct` within function ...




```
# start of show…() function
showStudent:
# show…()'s prologue
   …
# function body
   …
   # t0 = s.id
   lw   $t0, 4($fp)
   # t1 = s.prog
   lw   $t1, 48($fp)
   # print s.family
   la   $a0, 8($fp)
   li   $v0, 4
   syscall
   …
```

... Structs in MIPS

Can also pass a pointer to a `struct`

```
# Student stu;
# // set values in stu struct
# changeWAM(&stu, float newWAM);

   .data
stu: .space 56
wam: .space 4
   .text
   ...
   la   $a0, stu
   lw   $a1, wam
   jal  changeWAM
   ...
```

Clearly a more efficient way to pass a large `struct`

Also, required if the function needs to update the original `struct`

Exercise 3: Passing `structs` by reference

26/30

Write a MIPS function that implements:

```
typedef struct _Person {
   int  id_no;
   char family[15];
   char given[15];
} Person;

void showPerson(Person *p)
{
   printf("%d ", p->id_no);
   printf("%s, %s\n", p->family, p->given);
}
```

which might produce output like

```
5000035 Shepherd, John
```

Linked Structures in MIPS

27/30

C linked structures are typically composed of

- dynamically allocated `structs`
- where each `struct` holds pointer to other `struct`

Example:

```
typedef struct _node Node;
struct _node {
   int  value;  // value stored in Node
   Node *next;  // pointer to following Node
};
...
Node *first;
first = malloc(sizeof(Node));
first->value = 1;
first->next = malloc(sizeof(Node));
first->next->value = 2;
first->next->next = NULL;
```

... Linked Structures in MIPS

28/30

As noted above, SPIM doesn't have "proper" `malloc()`, but ...

```
   ...                # $s0 represents Node *first
   li   $a0, 8        # sizeof(Node) == 8
   jal  malloc
   move $s0, $v0      # s0 = malloc(sizeof(Node))
   li   $t0, 1
   sw   $t0, 0($s0)   # s0->value = 1
   li   $a0, 8        # required: $a0 not persistent
   jal  malloc
   mv   $t1, $v0      # s1 = malloc(sizeof(Node))
   sw   $t1, 4($s0)   # s0->next = s1
   li   $t0, 2
   sw   $t0, 0($t1)   # s1->value = 2
   sw   $0, 4($t1)    # s1->next = NULL
   ...
```

`$s0` gives persistent access to first `Node`, which links to second `Node`

Exercise 4: Iteration over a Linked List

29/30

Give a MIPS implementation for the following C function:

```
void showList(Node *L)
{
   Node *cur;
   for (cur = L; cur != NULL; cur = cur->next) {
      printf("%d", cur->value);
      if (cur->next != NULL) printf("%c",',');
   }
   printf("%c",'\n');
}
```

which would produce, for the list defined above

```
1,2
```

Use `$s0` to represent the `cur` variable.

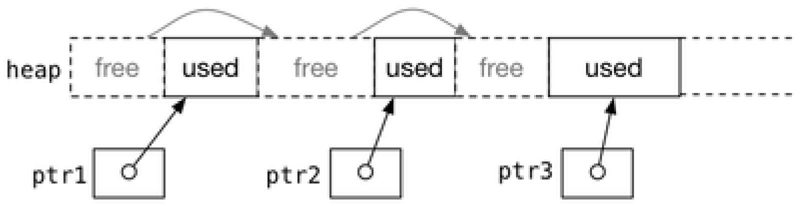
... Linked Structures in MIPS

30/30

SPIM does not have `free()`

Implementing C-like `malloc()` and `free()` in MIPS requires

- requires a complete implementation of C's heap management, i.e.
- a large region of memory to manage (`syscall` 9)
- ability to mark chunks of this region as "in use" (with size)
- ability to maintain list of free chunks
- ability to merge free chunks to prevent fragmentation



Produced: 1 Sep 2017