

Procedures and MMIO



Reminders:

Lab 1 due today, 2/14

Lab 1 checkoff due 2/20

Sign up for scheduled checkoff

Clarification: li, lui, and mv Instructions vs lw Instructions

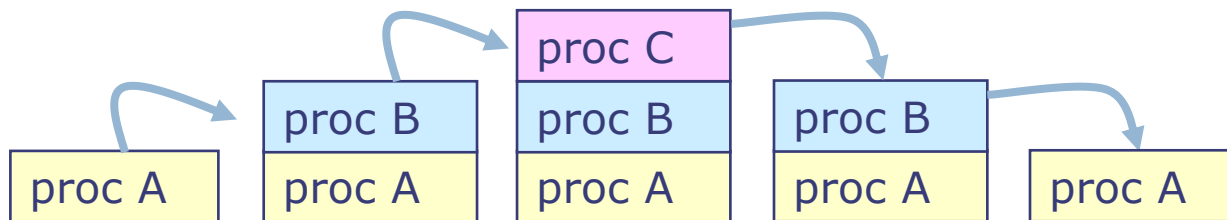
- li and lui are NOT memory operations
 - li – *load* immediate
 - li a1, 2 // sets a1 = 2
 - li a1, 0x12345 // sets a1 = 0x12345
 - lui – *load* upper immediate
 - lui a1, 2 // sets a1 = 0x2000
 - In both of these instructions the constants are encoded into the instruction. They are not loaded from memory.
- mv is NOT a memory operation
 - mv – move
 - mv a1, a0 // set a1 = a0
 - mv copies the contents of register a0 into register a1.

Clarification: lw and sw reference Memory

- lw and sw are memory operations
 - lw – load word – **load** from memory
 - lw a1, 0x200(x0) // sets a1 = Mem[x0 + 0x200]
 - sw – store word – store to memory
 - sw a1, 0x200(x0) // sets Mem[x0+0x200] = a1
- “Result is in a0. *Store* result in a2.”
 - Correct:
 - mv a2, a0
 - addi a2, a0, 0
 - add a2, a0, zero
 - Incorrect:
 - sw a0, 0(a2)

Recap: Activation frames and procedure calls

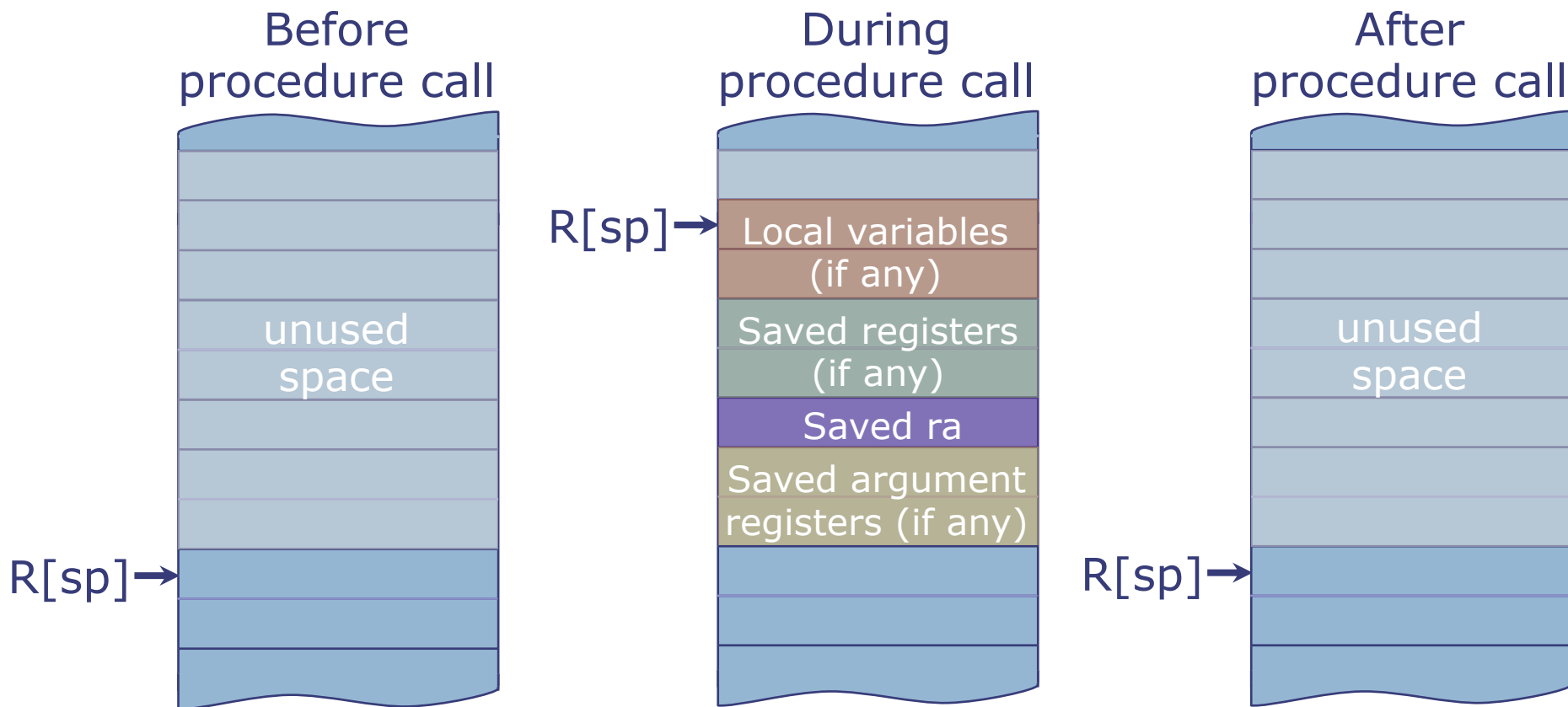
- An *Activation frame* holds a procedure call's data, i.e., arguments, return address, local or temporary variables
 - A new activation frame is allocated in memory when a procedure is called
 - An activation frame is deallocated at the time of the procedure exit
- Activation frames are allocated in a stack manner (Last-In-First-Out)



- The current procedure's activation frame is always at the top of the stack

Activation Frame Structure

- Compilers use a consistent stack frame convention:

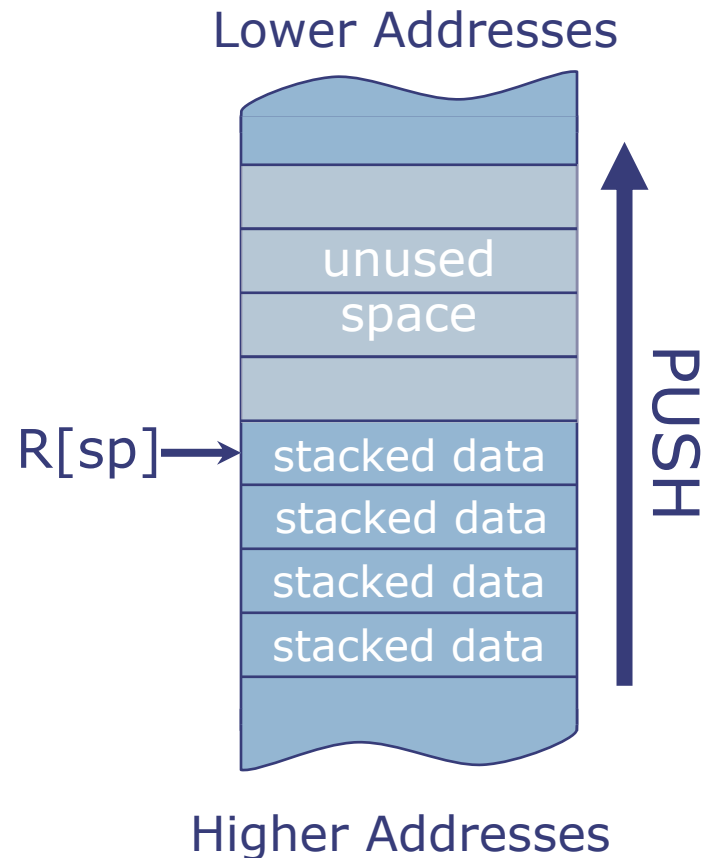


Recap: RISC-V Stack

- Stack is in memory
- Stack grows down from higher to lower addresses
- `sp` points to top of stack (last pushed element)
- Push sequence:

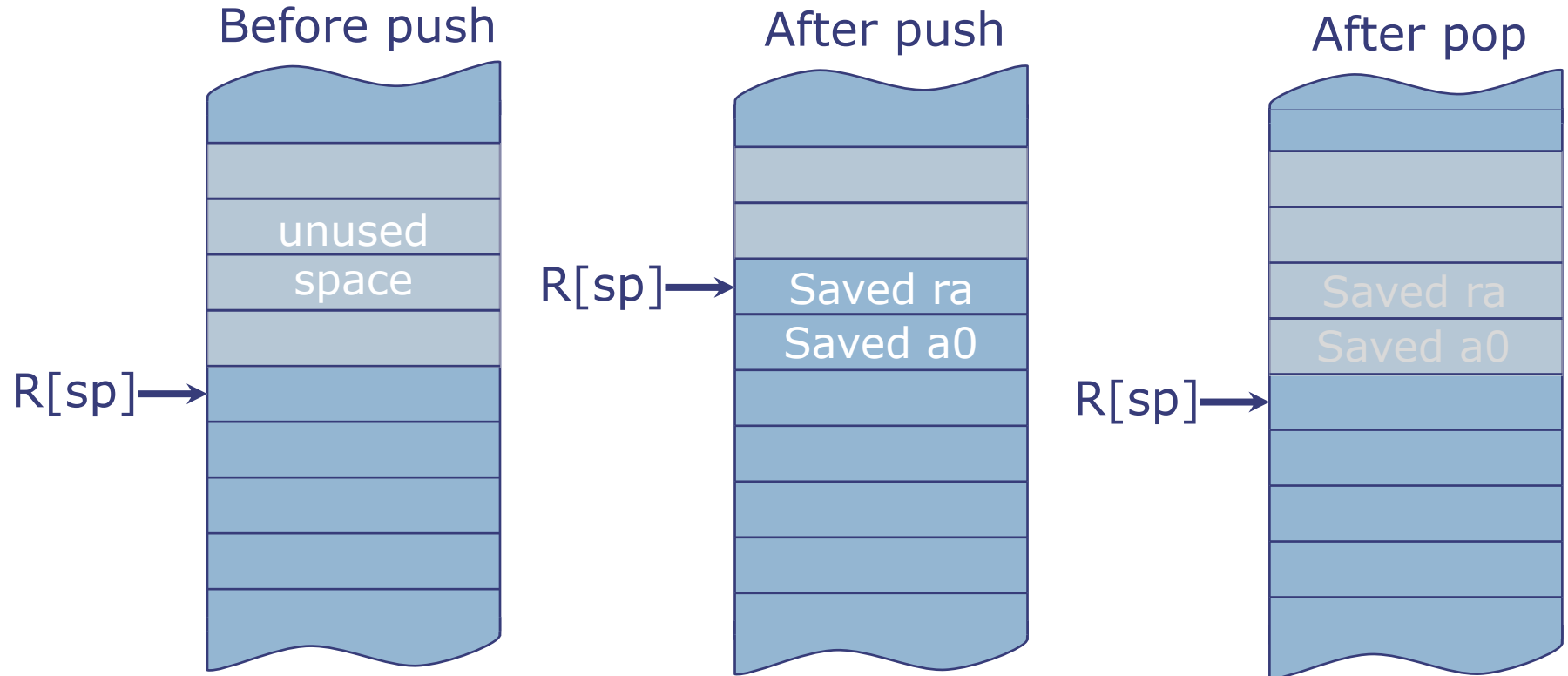
```
addi sp, sp, -4  
sw a1, 0(sp)
```
- Pop sequence:

```
lw a1, 0(sp)  
addi sp, sp, 4
```
- Discipline: Can use stack *at any time*, but leave it as you found it!



Recap: Using the stack

- Stack contents:



Sample push sequence

```
addi sp, sp, -8  
sw ra, 0(sp)  
sw a0, 4(sp)
```

Sample pop sequence

```
lw ra, 0(sp)  
lw a0, 4(sp)  
addi sp, sp, 8
```

Recap: RISC-V Registers

Symbolic name	Registers	Description	Saver
a0 to a7	x10 to x17	Function arguments	Caller
a0 and a1	x10 and x11	Function return values	Caller
ra	x1	Return address	Caller
t0 to t6	x5-7, x28-31	Temporaries	Caller
s0 to s11	x8-9, x18-27	Saved registers	Callee
sp	x2	Stack pointer	Callee
gp	x3	Global pointer	---
tp	x4	Thread pointer	---
zero	x0	Hardwired zero	---

Recap: Calling Conventions

Caller: Saves ra register on the stack prior to procedure call and restores it upon return. Also saves any aN or tN registers whose values need to be maintained past procedure call.

```
addi sp, sp, -8
sw ra, 0(sp)
sw a1, 4(sp)
call func
lw ra, 0(sp)
lw a1 4(sp)
addi sp, sp, 8
```

func:

Callee: Saves original value of sN registers before using them in a procedure. Must restore them before exiting procedure. You must also restore the sp to its original value.

```
func:
    addi sp, sp, -4
    sw s0, 0(sp)
    ...
    lw s0, 0(sp)
    addi sp, sp, 4
    ret
```

Nested Procedures

- If a procedure calls another procedure, it needs to save its own return address
 - Remember that `ra` is **caller**-saved
- Example:

```
bool coprimes(int a, int b) {  
    return gcd(a, b) == 1;  
}
```

`coprimes:`

```
    addi sp, sp, -4
```

```
    sw ra, 0(sp)
```

```
    call gcd
```

```
// overwrites ra
```

```
    addi a0, a0, -1
```

```
    sltiu a0, a0, 1
```

```
    lw ra, 0(sp)
```

```
    addi sp, sp, 4
```

```
    ret
```

```
// needs original ra
```

Recursive Procedures

- Recursive procedures are just one particular case of nested procedures
- Example:

```
// Computes nth Fibonacci number
// Assume n >= 0
int fib(int n) {
    if (n < 2) return n;
    else return fib(n-1) + fib(n-2);
}
```

Order of these instructions
is critical for correct behavior

```
fib:
    li t0, 2
    blt a0, t0, fib_done
    addi sp, sp, -8
    sw ra, 4(sp)
    sw s0, 0(sp)
    mv s0, a0 // save n
    addi a0, a0, -1
    call fib
    mv t0, s0 // t0 = n
    mv s0, a0 // save fib(n-1)
    addi a0, t0, -2
    call fib
    add a0, s0, a0
    lw s0, 0(sp)
    lw ra, 4(sp)
    addi sp, sp, 8
fib_done: // result in a0
    ret
```

Computing with large data structures

- Suppose we want to write a procedure `vadd(a, b, c)` to add two arrays `a` and `b` and store the result in array `c`
 - Assume the arrays are too large to be stored in registers
- We will bring the elements of `a` and `b`, one by one, into the registers and after adding them store the result back in memory
- Since `vadd` can be called with different arguments, how should the arguments be passed from the calling procedure to `vadd`?

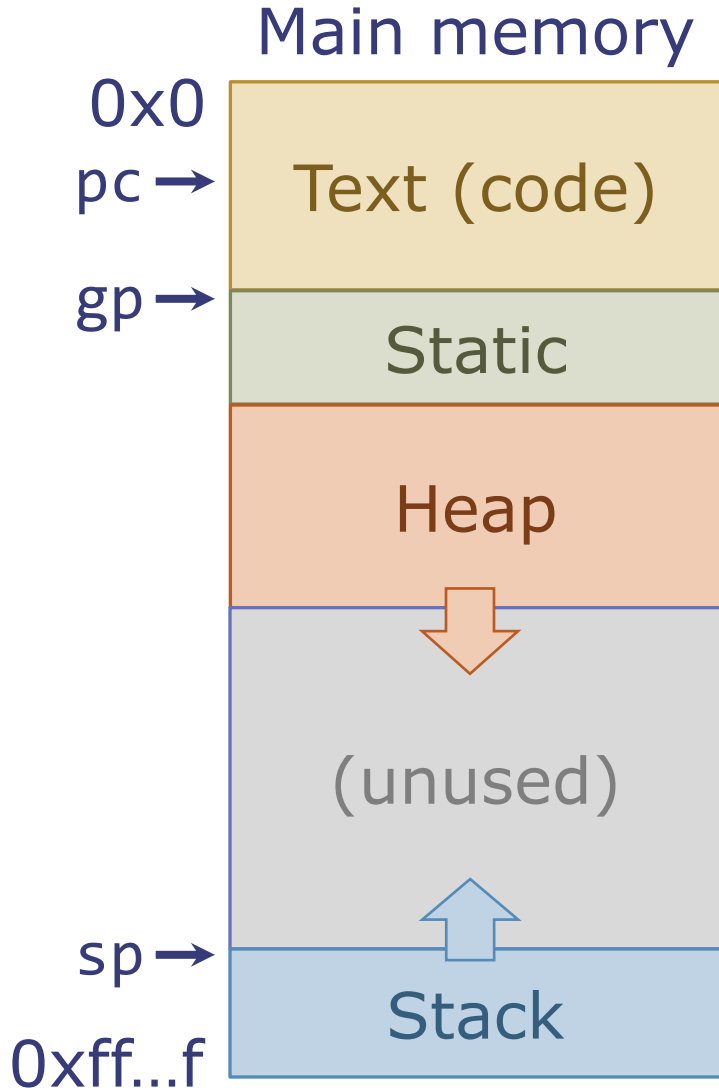
Passing arrays to a procedure

- We could divide up the arrays in chunks of size k , which are small enough to fit in the registers and then pass them to a `vadd` procedure that adds two small chunks together. Of course the caller will have to call this procedure n/k times to add all the elements!
- A better way is to pass the base address of `a`, `b` and `c` to the procedure and let it access the memory directly
 - This is quite different from what we have seen so far because a called procedure can access memory outside its activation frame on the stack

Memory Layout

- Most programming languages (including C) have three distinct memory regions for data:
 - **Stack**: Holds data used by procedure calls
 - **Static**: Holds global variables that exist for the entire lifetime of the program
 - **Heap**: Holds dynamically-allocated data
 - In C, programmers manage the heap manually, allocating new data using `malloc()` and releasing it with `free()`
 - In Python, Java, and most modern languages, the heap is managed automatically: programmers create new objects (e.g., `d = dict()` in Python), but the system frees them only when it is safe (no pointers in the program point to them)
- In addition, the **text region** holds program code

RISC-V Memory Layout



- Text, static, and heap regions are placed consecutively, starting from low addresses
- Heap grows towards higher addresses
- Stack starts on highest address, grows towards lower addresses
- sp (stack pointer) points to top of stack
- gp (global pointer) points to start of static region

Passing Complex Data Structures as Arguments

```
// Finds maximum element in an
// array with size elements
int maximum(int a[], int size)
{
    int max = 0;
    for (int i = 0; i < size;
        i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}

int main() {
    int ages[5] =
        {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```


Passing Complex Data Structures as Arguments

```
main:  li a0, ages
       li a1, 5
       call maximum
       // max returned in a0
```

```
ages:  23
       4
       6
       81
       16
```

```
int main() {
    int ages[5] =
        {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```

Passing Complex Data Structures as Arguments

```
// Finds maximum element in an
// array with size elements
int maximum(int a[], int size)
{
    int max = 0;
    for (int i = 0; i < size;
        i++) {
        if (a[i] > max) {
            max = a[i];
        }
    }
    return max;
}

int main() {
    int ages[5] =
        {23, 4, 6, 81, 16};
    int max = maximum(ages, 5);
}
```

```
maximum:
    mv t0, zero    // t0: i
    mv t1, zero    // t1: max
j compare
loop:
    slli t2, t0, 2 // t2: i*4
    // t3: addr of a[i]
    add t3, a0, t2
    lw t4, 0(t3)   // t4: a[i]
    ble t4, t1, endif
    mv t1, t4      // max = a[i]
endif:
    addi t0, t0, 1 // i++
compare:
    blt t0, a1, loop

    mv a0, t1      // a0 = max
    ret
```

Why not always use pointers as arguments?

// Find perimeter of a triangle

```
int perimA(int a, int b,
           int c) {
    int res = a + b + c;
    return res;
}
```

```
int perimB(int sides[], int
size) {
    int res = 0;
    for (int i = 0; i < size;
i++) {
        res = res + sides[i];
    }
    return res;
}
```

Indirection can be expensive

- Extra memory references
- Slower execution

perimA:

```
add t0, a0, zero // t0: res
add t0, t0, a1
add t0, t0, a2
mv a0, t0
ret
```

perimB:

```
mv t0, zero // t0: i
mv t1, zero // t1: res
j compare
loop:
    slli t2, t0, 2 // t2: i*4
    // t3: addr of sides[i]
    add t3, a0, t2
    lw t4, 0(t3) // t4: sides[i]
    add t1, t1, t4
    addi t0, t0, 1 // i++
compare:
    blt t0, a1, loop
    mv a0, t1 // a0 = res
    ret
```

Passing Complex Data Structures as Arguments or Return Values

- Other complex data structures like dictionaries, structures, linked lists, etc. would follow the same methodology of passing a pointer to the data structure as the argument to the procedure along with any additional required information like number of elements, etc.
- Similarly, when the return value is a complex data structure, then the data structure is stored in memory and a pointer to the data structure is returned by the procedure.

Take Home

- Discuss how one would implement a procedure that requires 10 arguments (i.e., they can't all fit in the argument registers).

Handling Inputs and Outputs

How do we handle Inputs and Outputs in Assembly?

- Memory Mapped I/O
 - Uses the same address space to map both memory and I/O Devices.
 - I/O Devices monitor the CPU memory requests and respond to memory requests that use the address associated with the I/O device.
 - MMIO addresses can only be used for I/O and not for regular storage.

MMIO Addresses

- Outputs:

- 0x 4000 0000 - performing a sw to this address prints an ASCII character to the console corresponding to the ASCII equivalent of the value stored at this address
- 0x 4000 0004 - a sw to this address prints a decimal number
- 0x 4000 0008 - a sw to this address prints a hexadecimal number

- Inputs

- 0x 4000 4000 - performing a lw from this address will read one signed word from the console.
- Repeating a lw to this address will read the next input word and so on.

Memory Mapped IO Example 1

```
// load the read port into t0  
li t0, 0x40004000
```

```
// read the first input  
lw a0, 0(t0)  
// read the second input  
lw a1, 0(t0)
```

```
// add them together  
add a0, a0, a1
```

```
// load the write port into t0  
li t0, 0x40000004  
// write the output  
sw a0, 0(t0)
```

MMIO for Performance Measures

- Performance Measures
 - 0x 4000 5000 – lw to get instruction count from start of program execution
 - 0x 4000 6000 – lw get performance counter – number of instructions between turning the performance counter on and then off.
 - 0x 4000 6004
 - sw 0 to turn performance counting off
 - sw 1 to turn it on

Memory Mapped IO Example 2

```
// prepare to read input from console
li t0, 0x40004000
// get user input
lw a0, 0(t0)
lw a1, 0(t0)
// load the performance counter address into t1
li t1, 0x40006000
li t2, 1
// start the performance counter by storing 1 to the magic address
sw t2, 4(t1)
add a0, a0, a1
// stop the performance counter by storing 0 to the address
sw zero, 4(t1)
// prepare to print decimal to console
li t0, 0x40000004
// first print sum
sw a0, 0(t0)
// get the count from the performance counter
lw t2, 0(t1)
// print the count
sw t2, 0(t0)
```

Thank you!

Next lecture: Digital Abstraction