What we (hope) to cover today

• Assembler Directives
  • Filling RAM (Bytes and Strings)
  • Using Labels
• Useful Constructs (If-Else, While)
• The Stack
• Functions
Assembly Language

- Remember this picture about our layers:
Assembly Language

• Assembly
  • ISA corresponds to what the microarchitecture can do
  • Still needs to go through an assembler to get to binary
    • Similar to how Python needs to go through a compiler
  • ISA + other features to make our lives easier = Assembly
Assembly Language

• A few definitions for Assembly
  • A line is an ISA operation, a label, or an assembler directive
    • We can have whitespace and comments, but they are ignored by the assembler
  • An ISA operation is any operation we have described, plus offsets can use label names, and instead of numbers, you are allowed to use characters
  • Comments are anything after a semicolon character
  • Assembler Directives
  • Labels
Assembler Directives
Assembler Directives

• Allow us to fill RAM with data before executing our program
  • Useful!
  • Two types:
    • Byte Directives
    • String Directives
Assembler Directives

• Byte Directive
  • Fills a section of RAM from a specified address with given values
  • Syntax:
    \[
    .byte(\text{address}) \text{ bytes, to, fill, separated, by, commas}
    \]

An example:

\[
.\text{byte}(1231) \ 12, 189, 200, 0, 0
\]

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1231</td>
<td>12</td>
</tr>
<tr>
<td>1232</td>
<td>189</td>
</tr>
<tr>
<td>1233</td>
<td>200</td>
</tr>
<tr>
<td>1234</td>
<td>0</td>
</tr>
<tr>
<td>1235</td>
<td>0</td>
</tr>
</tbody>
</table>
Assembler Directives

• Byte Directive
  • What’s another thing that is represented by a byte? Characters!
  • ASCII Characters all have a single byte value (Chapter 6)


<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>504</td>
<td>H</td>
</tr>
<tr>
<td>505</td>
<td>e</td>
</tr>
<tr>
<td>506</td>
<td>l</td>
</tr>
<tr>
<td>507</td>
<td>l</td>
</tr>
<tr>
<td>508</td>
<td>o</td>
</tr>
</tbody>
</table>
Assembler Directives

- Storing a bunch of characters like that can be clumsy
- String directive
  - Stores a string of characters (special characters escaped using \\)
  - Syntax:
    `.string(address) “string to store”`

Example:

`.string(504) “Hello”`

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>504</td>
<td>H</td>
</tr>
<tr>
<td>505</td>
<td>e</td>
</tr>
<tr>
<td>506</td>
<td>l</td>
</tr>
<tr>
<td>507</td>
<td>l</td>
</tr>
<tr>
<td>508</td>
<td>o</td>
</tr>
</tbody>
</table>
Labels
Labels

• Labels allow us to specify locations in our program, and use the label names instead of offsets

• Label names are constructed using numbers, letters, or underscores. They cannot begin with a number

• The label will then point to the following code after the label

• Syntax
  Label_name:
  [other code]
Useful Constructs (Using Labels)

• If/Else

```python
if(x == y):
    x = x + 1
else:
    y = y + 1
z = z + x + y
```

;Assembly

; x is in r30
; y is in r31
; z is in r29

cp r30, r31
breq if_block

else_block:
    inc r31 ; Increment y
rjmp after_block

if_block:
    inc r30 ; Increment x

after_block:
    add r29, r30 ; Add z and x
    add r29, r31 ; Add (z+x) and y
Useful Constructs (Using Labels)

• While

```python
while(x == y):
    x = x - 2
```

;Assembly
;x is in r30
;y is in r31

```assembly
begin_while:
    cp r30, r31
    brne done_while
    subi r30, 2
    rjmp begin_while

done_while:
```
The Stack
The Stack

• A way to save and restore values in registers
• Uses two operations, “pushing” and “popping”
• Works like a stack of plates
  • “Pushing” is like adding a plate on top of the stack
  • “Popping” is like taking the plate off the top of the stack
• Essential for Functions
The Stack

• Values are stored in decreasing addresses in memory.
• The top of the stack is tracked using the stack pointer (sp), a 16 bit value stored in IO registers.
The Stack

• Two instructions to operate on the stack
  • Pushing a value
    • Store the value at the stack pointer
    • Decrement the stack pointer
    Syntax:
      push [register to push value from]
  • Popping a value
    • Increment the stack pointer
    • Take the value from the stack pointer
    Syntax:
      pop [register to store popped value]
The Stack

• One final thing: sp is NOT set automatically!
  • Stack pointer corresponds to I/O registers 61 and 62.
  • IO[62] contains the upper 8 bits, IO[61] contains the lower 8 bits.
  • We need to set these registers before we use the stack.

Example:

\[5000 = 136 + 19 \times 256 \rightarrow\]

\[
\begin{align*}
    \text{ldi} & \ r31, \ 136 \\
    \text{out} & \ 61, \ r31 \\
    \text{ldi} & \ r31, \ 19 \\
    \text{out} & \ 62, \ r31
\end{align*}
\]
The Stack

Example:

```
push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28
```
The Stack

Example:

- push r29
- push r30
- push r27
- pop r30
- push r30
- push r28
- pop r31
- pop r28

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>255</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td></td>
</tr>
<tr>
<td>4999</td>
<td></td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>
The Stack

Example:

push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>255</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td>255</td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>
The Stack

Example:

push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>255</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td>4</td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

William Galliher - CS 252 - UW Madison
The Stack

Example:

push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28

**Register File**

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

**RAM**

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td></td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

**sp**

4998
The Stack

Example:

```
push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td>4</td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

William Galliher - CS 252 - UW Madison
The Stack

Example:

```
push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>120</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td>2</td>
</tr>
<tr>
<td>4998</td>
<td>4</td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

William Galliher - CS 252 - UW Madison
The Stack

Example:

push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sp</th>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4997</td>
<td>4996</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4997</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4998</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td></td>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

William Galliher - CS 252 - UW Madison
The Stack

Example:

```
push r29
push r30
push r27
pop r30
push r30
push r28
pop r31
pop r28
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>27</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>4</td>
</tr>
<tr>
<td>29</td>
<td>3</td>
</tr>
<tr>
<td>30</td>
<td>4</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4996</td>
<td></td>
</tr>
<tr>
<td>4997</td>
<td></td>
</tr>
<tr>
<td>4998</td>
<td></td>
</tr>
<tr>
<td>4999</td>
<td>255</td>
</tr>
<tr>
<td>5000</td>
<td>3</td>
</tr>
</tbody>
</table>

`sp` is located at 4998.
Functions
Functions

• We want functions to operate similar to Python, where we call them, and the function comes back when it is done.
• First, a few design decisions need to be considered
Functions

• Consideration: We only want functions to execute when we call them.

• Consideration: Programs execute by incrementing the PC counter (or modifying it in some way).

• Where should we put our functions in RAM?
  • We want functions to be before our program in memory (in a lower address of RAM).
  • We will jump around the functions we write to our program
Functions

Example, in assembly:

rjmp program
our_function:
... [function code]
program:
... [program code]
Functions

- Consideration: Functions will (probably) use registers for computation.
- Consideration: We don’t want to lose values we have stored in registers.
- What should we do?
  - We have something we can use to save registers, the stack!
  - Someone will need to save the registers on the stack
Caller Saved vs Callee Saved

• Caller Saved registers
  • The section of code that calls the function saves the registers it wants to save.
  • Requires either knowledge of which registers the function will modify, or having to save every register we might want to prevent being modified.
  • Functions are smaller and faster, since they don’t have to consider saving registers.
Caller Saved vs Callee Saved

Caller saved, in assembly:

rjmp program
;The function modifies r29, let’s show caller-saved registers
our_function:
... [function code]
program:
...
push r29
[call the function]
pop r29
Caller Saved vs Callee Saved

- Callee Saved registers
  - The function saves all values in registers it is going to modify, then restores them.
  - The code that calls the function doesn’t have to worry about saving these registers.
  - No unneeded saving happens
  - The function size is larger, meaning slower code.
Caller Saved vs Callee Saved

Callee saved, in assembly:

rjmp program
;The function modifies r29, let’s show callee-saved registers
our_function:
push r29
... [function code]
pop r29
program:
...
[call the function]
Functions

• One last thing to consider: instructions for calling functions and returning.
  • `rcall [offset]` and `ret`
  • These functions automatically push the PC + 1 to the stack and pops the value back whenever `ret` is called.
Functions

rjmp program
;The function modifies r29, callee-saved
our_function:
push r29
... [function code]
pop r29
ret
program:
...
rcall our_function
• AVR includes a mult instruction, but in many cheaper AVR chips, the mult instruction is not implemented.

• Let’s implement a function doing multiplication!
  • We want to use the stack and the function design considerations.
  • How should we do it?
    • Think about multiply like adding over and over
• Python code

def mult(x,y):
    iteration = 0
    result = 0
    while(iteration != y):
        result = result + x
        iteration = iteration + 1
    return result

mult(4,5)
;Initialize sp
ldi r31, 136
out 61, r31
ldi r31, 19
out 62, r31
rjmp program
;multiply assumes the values desired are in r20, r21.
;The result is put in r22. r19 is modified, and callee-saved
multiply:
push r19
ldi r19, 0
ldi r22, 0
begin_mult:
cp r19, r20
breq done_mult
add r22, r21
inc r19
rjmp begin_mult
done_mult:
pop r19
ret
program:
ldi r20, 4 ;Initialize r20
ldi r21, 5 ;initialize r21
push r22 ;Caller-Save r22, where the result will be saved
rcall multiply
...
Summary

• Assembly Language
  • Assembled with an assembler, like a programming language is compiled with a compiler
  • Has neat tricks, like Assembler Directives and Labels

• Stack
  • Uses the stack pointer, which is in IO[61] and IO[62]
  • Operates using push and pop
  • Starts at initial sp and stores in decreasing addresses in RAM
Summary

• Functions
  • Functions should be put before program code and jumped around, so they are not executed accidentally.
  • Modified registers need to be saved
    • Caller vs Callee Saved
  • Uses `ret` and `rcall`

William Galliher - CS 252 - UW Madison
Outline

- Assembly Directives (filling RAM, labels)
- Useful constructs (if-else, while)
- Stack
- Functions

Assembly Language

In the ISA of the layers stack
ISA corresponds to what the microarchitecture can do
Still need to go through an assembler to get to binary
ISA + other features
A line of assembly language contains ISA operation, a label, or an assembler directive
  - ISA operation: add, load, compare
  - labels: a location specification
  - directives: instead of numbers, we can use characters to fill RAM
  - comments: anything after semicolon to annotate code

Assembler directives
  - Allows us to fill RAM with data before executing program
  - Byte direction
    - fills a section of RAM from a specified address with given values
      .byte(address) bytes,to,fill,separated,by,commas
    - Can also be used to represent ASCII characters
  - String directive
    - stores a string of characters in RAM
      .string(address) “string to store”
      .string(504) “Hello”

Labels
  - allows use to specify locations in our program, and use the label names instead of offsets
  - Syntax:
    - label_name:
      [other code]

If-else
  # python
if(x==y):
    x = x + 1
else:
    y = y + 1
z = z + x + y

; assembly
; x is in r30
; y is in r31
; z is in r29
cp r30, r31
breq if_block

else_block:
    inc r31
    ; increment y
rjmp after_block

if_block:
    inc r30
    ; increment x

after_block:
    add r29, r30
    ; add x to z; z = z + x
    add r29, r31
    ; add y to z; z = z + x + y

While
    while(x == y):
        x = x - 2

; assembly
; x is in r30
; y is in r31

begin_while:
cp r30, r31
brne done_while
subi r30, 2
rjmp begin_while

done_while:

Stack
A way to save and restore values in registers
push to add element on top of stack
pop to take element off top of stack
values are stored in decreasing address in memory
the stack pointer (sp) points to the top of stack
stack is at the bottom to avoid overwriting our program code or heap

In assembly
pushing a value
  store the value at the stack pointer
  decrement the stack pointer
syntax:
    push [register to push value from]
popping a value
  increment the stack pointer
  take the value from the stack pointer
syntax:
    pop [register to store popped value]
The stack pointer is NOT set automatically!
  stack pointer corresponds to IO registers 61 and 62

pointer setup example
  5000 = 136 + 19*256

    ; assembly
    ldi r31, 136
    out 61, r31
    ldi r31, 19
    out 62, r31

stack example [see slides for full example]
  ; stack pointer is initially 5000
    push r29  ; push value of r29 to 5000
    push r30  ; push value of r30 to 4999
    push r27  ; push value of r27 to 4998
    pop r30   ; pop data in 4998 into r30
    push r30  ; push value of r30 to 4998
    push r28  ; push value of r28 to 4997
    pop r31   ;
    pop r28   ;

Functions
Want functions to operate similar to Python, where call them and get a return value
We want functions to execute only when we call them. Therefore, we want functions at
the beginning of our program so that the PC does not accidentally fall into it.
Assembly Example

rjmp program
our_function:
[code here to implement our function]
program:
[code here to implement program]

We need to pass values into functions.

Caller saved registers
the function caller saves the registers it wants to save
push values before calling the function
pop values after returning from the function

Callee saved registers
The function saves and restore all registers it is going to modify
push values at the beginning of the function
pop values at the end of the function

Instructions for calling functions
rcall [offset]
pushes PC register + 1 on stack
ret
pops stack and store to PC

Full Example of Multiplication function in AVR

# python code

def mult(x,y):
    iteration = 0
    result = 0
    while(iteration != y):
        result = result + x
        iteration = iteration + 1
    return result

mult(4,5)

; assembly
; we want to use the stack for function call
; prepare stack
ldi r31, 136
out 61, r31
ldi r31, 19
out 62, r31
; [see slide for full multiplication example]