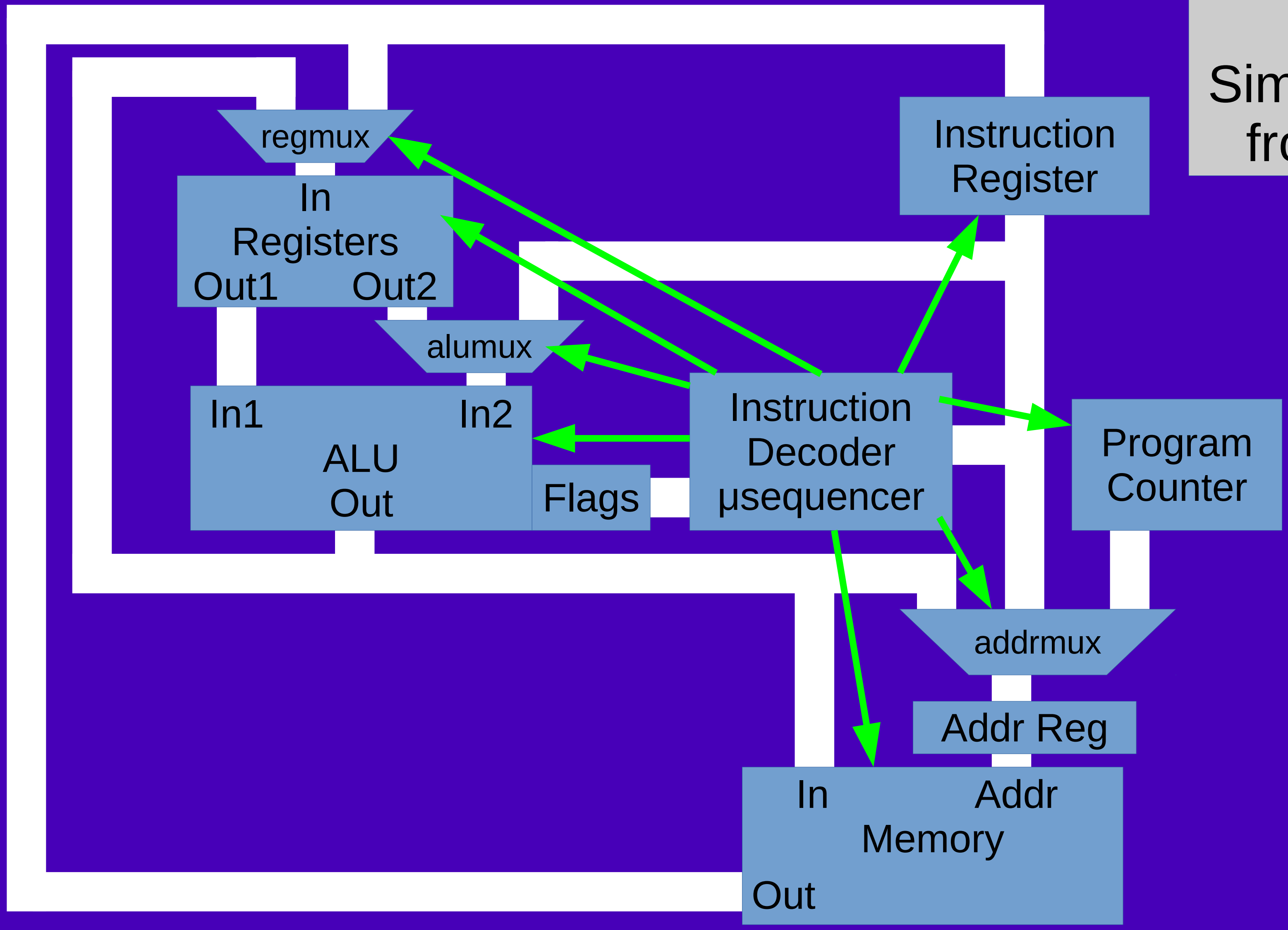
Microprocessor Overview

ECE 362
https://engineering.purdue.edu/ece362/

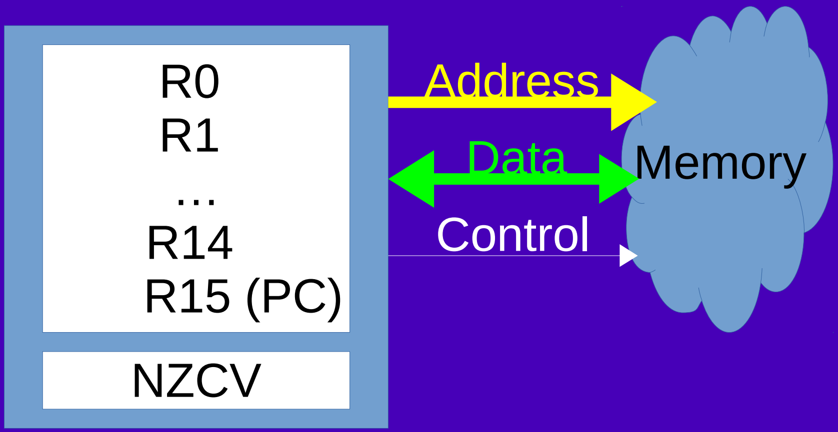
Reading Assignment

- Course Policies and Procedures (web page)
- Lab Policies and Procedures (web page)
- Textbook: Chapter 1, “See a Program Running”, pages 1 – 26
- Textbook: Chapter 2, “Data Representation”, pages 27 – 54. ***Just skim it***. You’ve seen all that in ECE 270, except for Section 2.5, “Character Strings”

Recall the Simple Computer from ECE 270



Condensed Simple Computer



Instructions:

Instr.	Mnemonic		Description	Flags
0rnn	ORI	Rr,#nn	Rr = (Rr) | nn	NZ
1rnn	ANDI	Rr,#nn	Rr = (Rr) & nn	NZ
2rnn	BICI	Rr,#nn	Rr = (Rr) & ~nn	NZ
3rnn	ADDI	Rr,#nn	Rr = (Rr) + nn	NZCV
4rnn	SUBI	Rr,#nn	Rr = (Rr) - nn	NZCV
5rnn	CMPI	Rr,#nn	(Rr) - nn	NZCV
6rnn	LDIBU	Rr,#nn	Rr = nn	NZ
…				
ar00 xxxx	LDL	Rr,xxxx	Rr = mem[xxxx]	
cr00 xxxx	STL	Rr,xxxx	mem[xxxx] = (Rr)	

Example program:

```
Addr.   Value       Mnemonic
0000    a100 0020   LDL  R1,0020   // Load R1 from memory 0020
0004    3105        ADDI R1,#05    // Add 5 to R1
0006    a300 0024   LDL  R3,0024   // Load R3 from memory 0024
000a    4303        SUBI R3,#03    // Subtract 3 from R3
000c    c300 002c   STL  R3,0028   // Store R3 to memory 0028
0010    c100 0028   STL  R1,002c   // Store R1 to memory 002c
…
0020    af35 9de2
0024    b3ee c9a5
0028    0000 0000
002c    0000 0000
```

What is the final result stored in 0028?

What is an Assembler?

- Software that converts symbolic expressions into raw bits.
  - LDL R1,0020 ==> a100 0020
- *Abstracts* memory locations into symbols
  - These are called labels
  - If we created a label called “var” the assembler might assign it to address 0020, and it would allow us to say: LDL R1,var ...and the assembler would turn it into... a100 0020
  - A label is a memory location, AKA an address
- Does the same thing with numerical constants.

What if we had a C compiler?

(And an assembler.)

C code:

```
int x = 2;
int y = 6;
int z = 3;

void main() {
    x = y - z;
}
```

compile

Assembly code:

```
.text
main:
      LDL R0,y
      LDL R1,z
      SUB R0,R1
      STL R0,x
.data
x:
.word 2
y:
.word 6
z:
.word 3
```

Read-only data follow

Label: assign an address

Writable data follow

Alloc space And initialize

Items highlighted in red are called assembler directives

The assembler assigns the following addresses:

```
main:   0000
x:      0020
y:      0024
z:      0028
```

assemble

Machine code:

```
0000  LDL R0,0024    // main
0004  LDL R1,0028
0008  SUB R0,R1
000a  STL R0,0020
…
0020  0000 0002      // x
0024  0000 0006      // y
0028  0000 0003      // z
```

How about loops?

```
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        x = x + y;
        y = y - z;
    } while (y != 0);
}
```

The simple computer was extended with the BNE instruction for this purpose.

BNE checks the Z flag and *branches* if it is not set.

(It Branches if result Not Equal to zero.)

```
.text
main: ldl  r0,z   // x = z;
      stl  r0,x
L0:   ldl  r0,x   // x = x + y
      ldl  r1,y
      add  r0,r1
      stl  r0,x
      ldl  r0,y   // y = y - z
      ldl  r1,z
      sub  r0,r1
      stl  r0,y
      ldl  r0,y   // while y != 0
      cmpi r0,#0
      bne  L0
.data
x: .word 7
y: .word 6
z: .word 1
```

How about function calls?

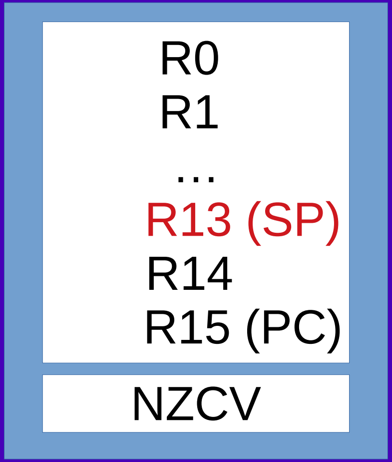
```
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        more();
        y = y - z;
    } while (y != 0);
}

void more() {
    x = x + z;
    return;
}
```

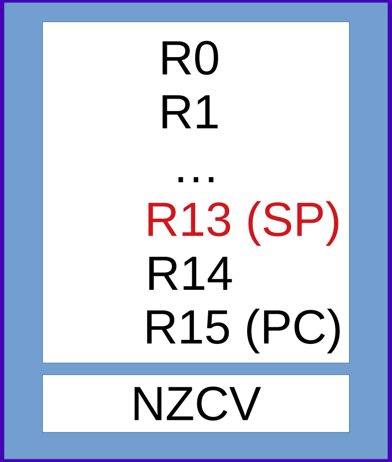
- We need something to "remember" where we left off when we called a 2nd function.
- We might want to call a 3rd function inside the 2nd function.
  - We might even want to call a function from itself. (Recursion)
    - We need an "unlimited" number of places to remember where we left off.
      - Because we don't know how "deep" calls will be.
        - Functions could call each other indefinitely!

Use R13 as a "stack pointer"



- SP starts at high address.
- A "JSR xxxx" instruction does the following:
  - Decrement SP.
  - Put the PC value in the memory location pointed to by the new SP.
  - Jump to the address xxxx.
- This is called "Jump to Subroutine"

Use R13 as a "stack pointer"



- An "RTS" instruction does the following:
  - Get the value of the memory location pointed to by SP and places it in PC.
  - Increment SP.
- RTS “undoes” everything JSR does.

How about function calls?

```
int x = 7;
int y = 6;
int z = 1;

void main() {
    x = z;
    do {
        more();
        y = y - z;
    } while (y != 0);
}

void more() {
    x = x + z;
    return;
}
```

```
.text
main: ldl  r0,z    // x = z;
      stl  r0,x
L0:   jsr  more    // more()
      ldl  r0,y    // y = y - z
      ldl  r1,z
      sub  r0,r1
      stl  r0,y
      ldl  r0,y    // while y != 0
      cmpi r0,#0
      bne  L0
      hlt
more: ldl  r0,x    // x = x + z
      ldl  r1,z
      add  r0,r1
      sta  r0,x
      rts          // return
.data
x: .word 7
y: .word 6
z: .word 1
```

How many memory words are needed to hold this program?

32 bits (4 bytes) for LDL, STL, JSR.
16 bits (2 bytes) for other instructions.
32 bits (4 bytes) for each word of data.

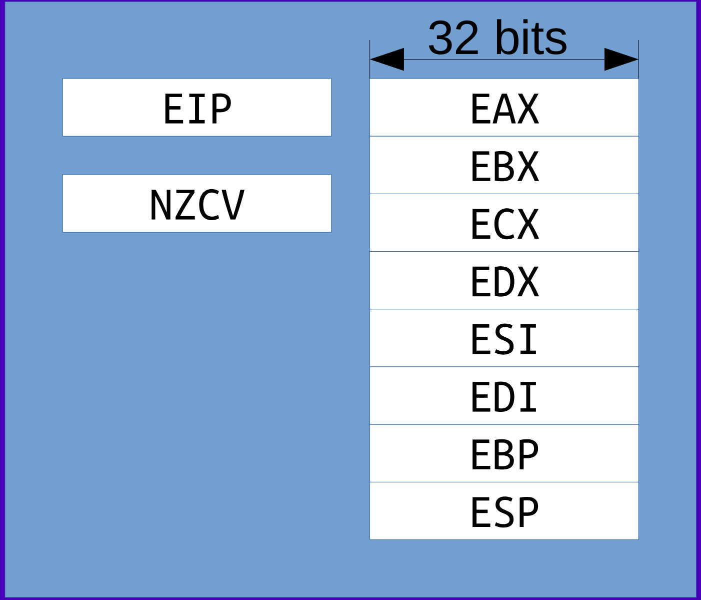
52 bytes of instruction
12 bytes of writable data

Other stack operations

- Simple computer was also extended to support push (PSH) and pop (POP).
- It’s too hard to demonstrate those with C.

Why look at the simple computer?

- Once you understand assembly language for one machine, you have no trouble with any other.



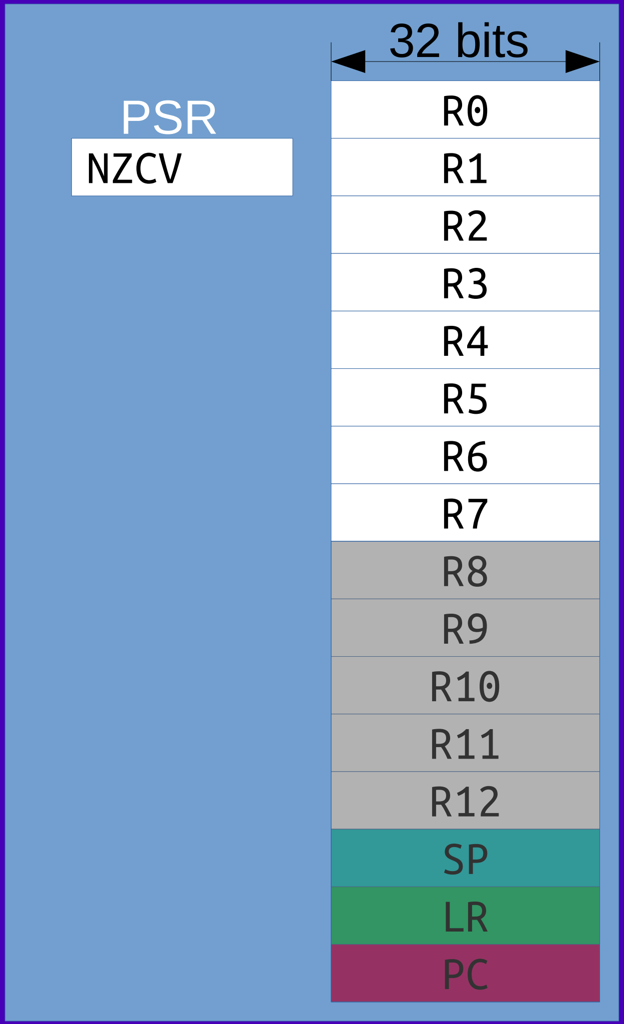
This is the 32-bit x86 architecture.

```
.data
x:    .long 7
y:    .long 6
z:    .long 1
.text
more:
      movl x, %ecx
      movl z, %edx
      addl %ecx, %edx
      movl %edx, x
      ret
main:
      movl z, %eax
      movl %eax, z
.L4:
      call more
      movl y, %edx
      movl z, %eax
      subl %eax, %edx
      movl %edx, y
      movl y, %eax
      testl %eax, %eax
      jne .L4
      ret
```

The Computer For This Course

- The computer we'll be studying in this course is the ARM Cortex-M0
  - A simple CPU with a minimal instruction set
  - Students sometimes think that, since their cellphone or their Raspberry Pi contain an ARM CPU, they will learn how to program it… no
  - The ARM CPU in other devices is very different from the ARM Cortex-M0.

Architecture



- Little Endian. Just like Intel x86.
- 16 registers. Looks simple.
  - R15 is the PC.
  - R13 is the SP.
  - R14 is the "Link Register" (LR)
  - Most instructions can only use R0 – R7.
- Program Status Register.
  - Upper four bits are the flags.

Not a discrete CPU

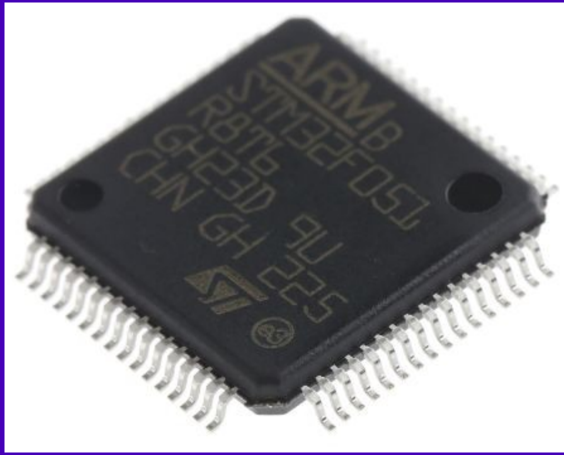
- This course formerly used a single standalone CPU which was connected to a separate RAM, ROM, and peripherals.
- Simple CPUs no longer exist all by themselves. They are bundled together with all of those extra things, all in one chip, called a "microcontroller."

We’ll use a development board

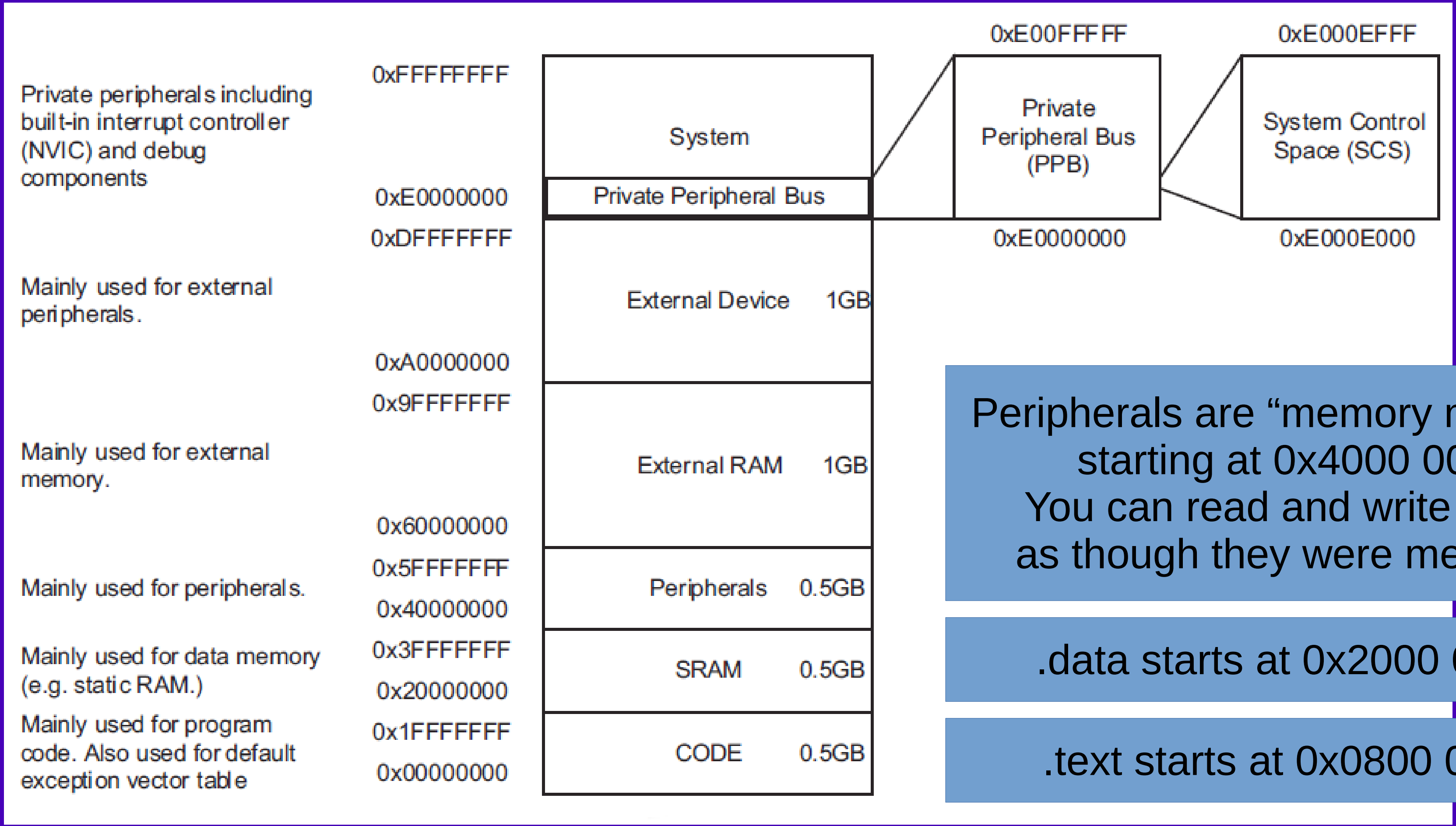
- It’s usually too hard to use a bare microcontroller
  - Solder the LQFP64 package
  - Add all the support electronics
- Development board provides an easier start.
- Finished product will use the raw chip.
- There are two qualities of an “embedded” CPU.

Embedded CPU has lots of gadgets

- 32-bit ARM Cortex-M0 CPU.
- 8MHz clock multiplied x6 by a phase-locked loop (PLL) to get 48 MHz.
- 3-stage pipeline. Most instructions take 3 cycles, but a new instruction starts each cycle.
- Built-in Flash ROM to store program and SRAM to hold read/write data.
- 32 maskable interrupt channels w/ 4 priority levels
- 12-bit ADC, 12-bit DAC
- SPI, I^2S, I^2C, USART, CAN, HDMI CEC peripherals
- Several timers with integrated PWM
- So much more:
  - https://engineering.purdue.edu/ece362/refs/



Memory Model



Peripherals are “memory mapped” starting at 0x4000 0000
You can read and write them as though they were memory.

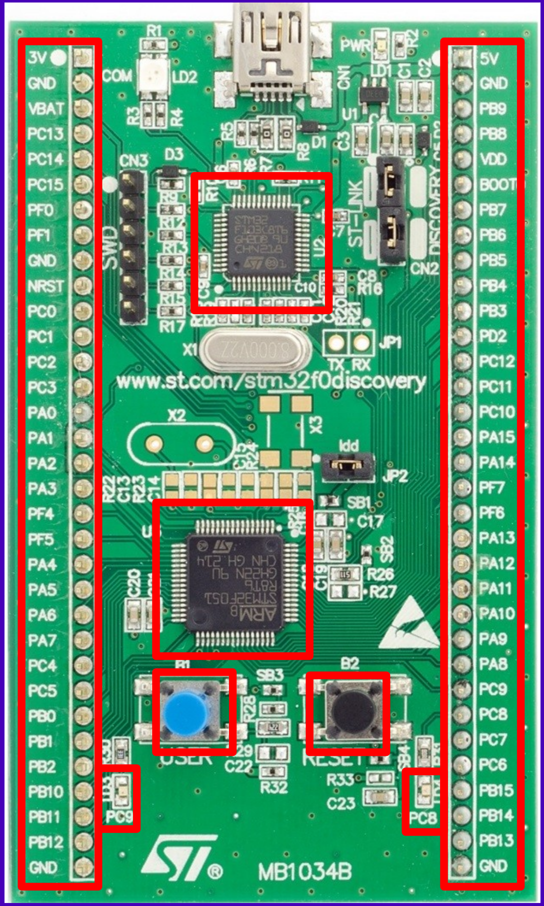
.data starts at 0x2000 0000

.text starts at 0x0800 0000

Embedded CPU Is Low Power

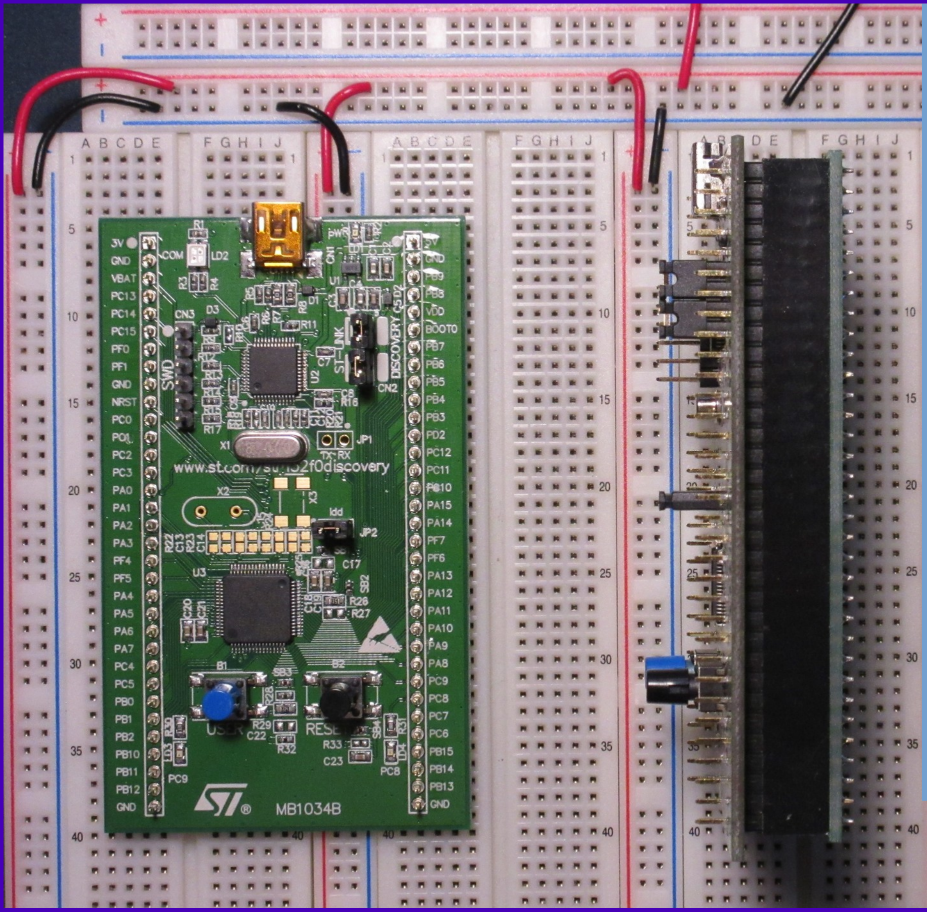
- Maximum power dissipation: 400 – 500mW
- No heatsink needed. Easy to “embed.”
- Sometimes people try to embed a conventional high-performance CPU.
  - This kind of thing needs a heatsink, fan, ventilation, etc.
- By comparison, the old laptop that I use for lectures has an Intel Core i7-3720QM. Up to 45W TDP!
  - Not good for embedded use.

(Old) Dev system for ECE 362



- STM32F0-Discovery board
- Contains STM32F051R8T6.
- Integrated programmer.
- Headers for LQFP64 I/O pins.
- 2 CPU-controllable LEDs.
- 2 push buttons (one is reset)

Discovery Board Limitations

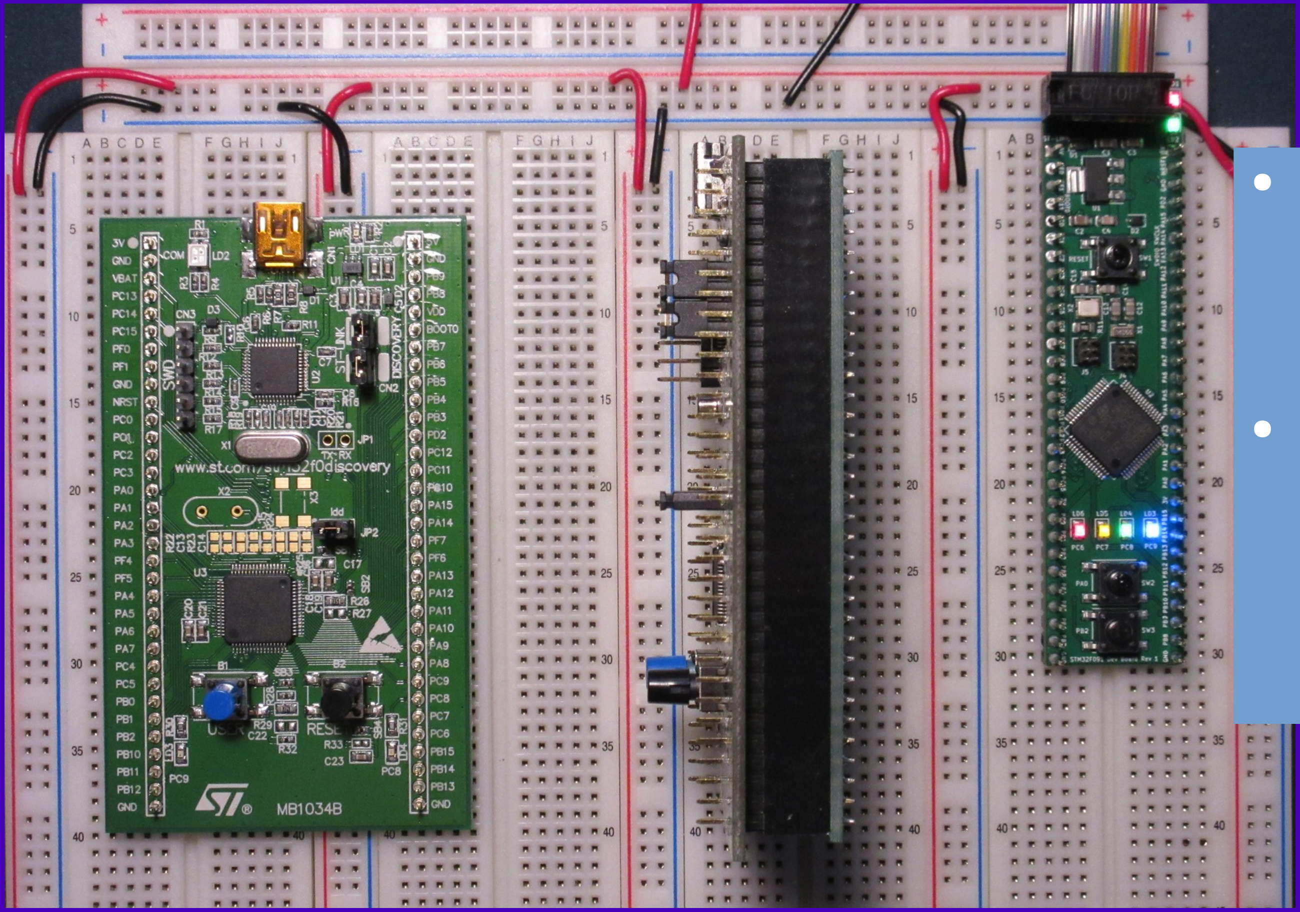


- Too wide for a breadboard
- Top pins too short to hold wires
- Pins in arbitrary order
- 5V/3V outputs next to GND
  - Shorting them will smoke a diode
- Confusing on-board programmer
- STM32F051R8T6 has only
  8K of SRAM, 64K of Flash ROM

Let’s Use a Better Micro

- How about the STM32F091RCT6?
  - It’s still a Cortex-M0 CPU
    - … so it’s just as easy to understand
  - The peripherals are similar to the STM32F051R8T6
  - 32K of SRAM and 256K of Flash ROM
  - But the only development board is the Nucleo64, and that has some very "interesting" problems that no student should ever experience.

Design A Development Board

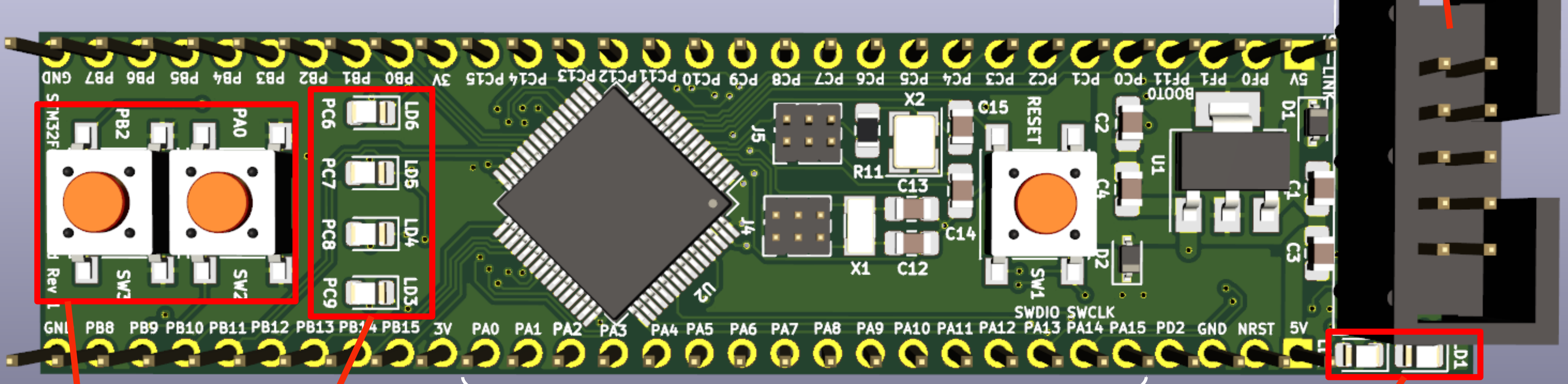


- Corrected the problems with the Discovery board
- Uses an STM32F091RCT6
  - 32K SRAM
  - 256K Flash ROM

New Development Board

External programmer

Reference design available for students to use to create their own.



2 "user" buttons

4 LEDs (RYGB)

Common pins grouped together
(All pins long enough to hold wire connectors)

2 power LEDs
(5V / 3V)

Every system is different

- Every system is similar
- Every concept we learn about in this class should be applicable to any microcontroller
  - Unfortunately, there often will be lots of specific things to say about the STM32F091RCT6

Next topic

- The ARM Cortex-M0 Instruction Set