Administrivia

Lab 2 due Friday @5:00

- Demo to Mark or me
  - All group members must be present
  - 4 different courses; count best 3 tries out of 4
- Group report: only turn in one copy for group
- Personal report: raw text format
- Late policy: 10%/day (up to 2 days)

Today

- Serial communication
- Lab 3
- Atmel mega8 architecture

Digital Communication

Transmission of digital data between computing elements, e.g. :

- Components within a processor
- Distinct processors

Digital Communication

So far: we have largely seen **parallel** data transfers

- N bits are transmitted simultaneously along N signal lines
- N is typically a multiple of 8

- This approach can transfer a lot of data quickly, but it has a disadvantage …

Parallel Communication

Disadvantage:

- Requires many signal lines
- This is particularly impractical as the distance between the two devices increases

What is the alternative?

Digital Communication

What is the alternative?

- Transmit the data in smaller “chunks”
- And then multiplex the different chunks in time
  - We call this **serializing**
- This requires a smaller number of signal lines

Serial Communication

Extreme case:

- Only send one bit at a time

- How the bits are ordered depends on the protocol specification
- Requires more time (than the parallel case) to send the data
- But: we need a very small number of signal lines

Translating a Bit into a Signal on a Wire

How do we do this?

Translating a Bit into a Signal on a Wire

How do we do this?

- Hold the line low or high depending upon the value of the bit
- We do this for some amount of time

Challenge:

- How does the receiver know when to read the bit value from the line?

Translating a Bit into a Signal on a Wire

How does the receiver know when to read the bit value from the line?

- We must have some way of agreeing when the line value is valid

Translating a Bit into a Signal on a Wire

When is the line value valid?

- In **synchronous serial** approaches:
  - Another signal line carries a **clock signal**
  - The two sides agree that when the clock transitions, the bit value is valid
  - Typically, one side generates the clock signal

Asynchronous Serial

In asynchronous serial:

- There is no shared clock
- Instead, both sides have their own clock

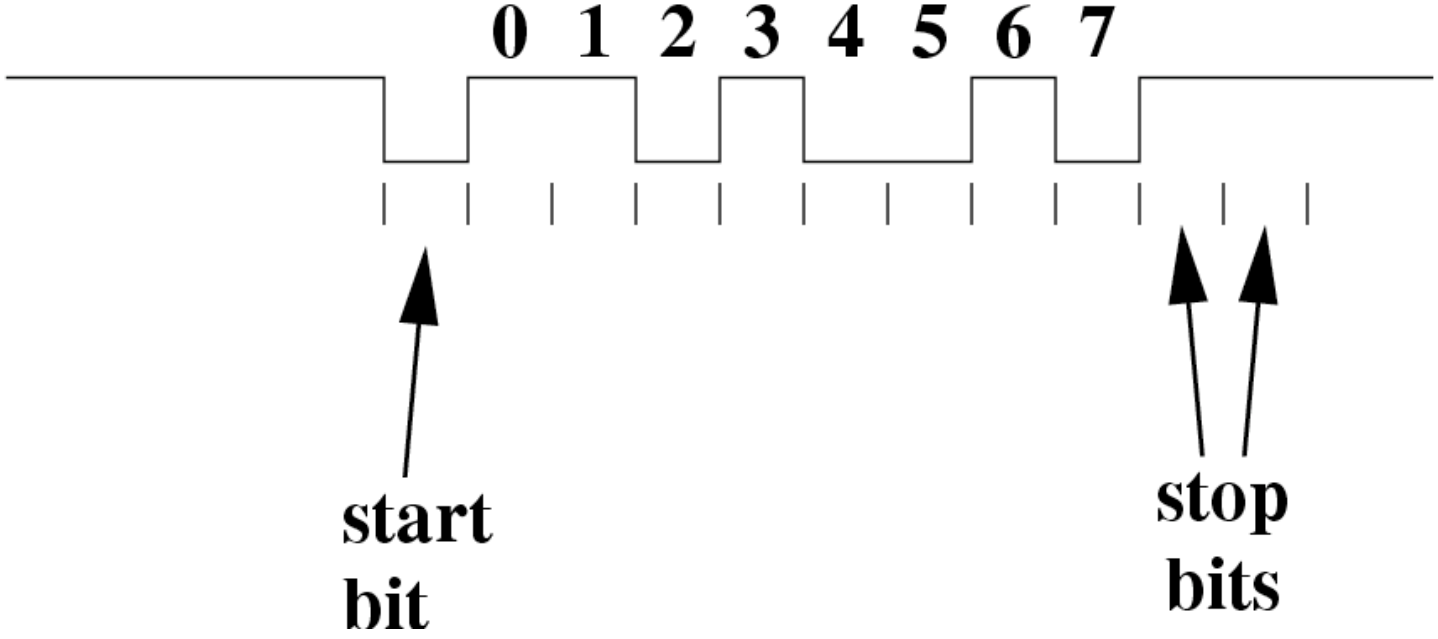
- This removes the need for a clock signal line (good)
- But how do the two sides agree about the time that the data is valid?

Asynchronous Serial Communication

How can the two sides agree that the data is valid?

- Must both be operating at essentially the same transmit/receive frequency
- A data byte is prefaced with a bit of information that tells the receiver that data is coming
- The receiver uses the arrival time of this **start bit** to synchronize its clock

A Typical Data Frame



The stop bits allow the receiver to immediately check whether this is a valid frame

- If not, the byte is thrown away

Data Frame Handling

Most of the time, we do not personally deal with the data frame level. Instead, we rely on hardware solution:

- Universal Asynchronous Receiver Transmitter (UART)
  - Very common in computing devices (even the mega8 devices)

Frame-Level Error Detection

- Due to timing and noise problems, the receiver may not receive the correct data
- We would like to catch these errors as early in the process as possible
- The first line of defense: include extra bits in the data frame that can be used for error detection and/or correction
  - This can be done by our UART

Frame-Level Error Detection

Parity bit: indicates whether there is an odd or even number of 0s in the byte

- Transmitter computes the parity bit and includes it in the data frame
- Receiver also computes parity of the received byte
- If the two do not match, then an error is raised

Frame-Level Error Correction

- When we use a single parity bit, we assume that in the worst case, a single bit is corrupted
- But: we can be more sophisticated about catching errors if we transmit more bits (at the cost of a larger data frame)
- Instead, we tend to do these types of checks on a set of bytes (an example in a few slides)

One Asynchronous Serial Standard: RS232-C

Defines a logic encoding standard:

- “High” is encoded with a voltage of -5 to -15 (-12 to -13V is typical)
- “Low” is encoded with a voltage of 5 to 15 (12 to 13V is typical)

This is what the AVR ISP devices speak!

Lab 3

- Instead of wires, we will use infrared communication
- Our beacons will no longer transmit a constant signal
- Instead, we will modulate the 40KHz carrier in order to encode data

Lab 3

- The data packet will tell us two things:
  - The ID # of the beacon
  - The ID # of another beacon

- Problem:
  - Construct a circuit (using a mega8) that will receive the infrared signal (tied directly to one of the sensors)
  - Display the incoming data with a set of LEDs

Lab 3: Packet Specification

A valid packet will be composed of a specific sequence of bytes:

- 0x4F (ASCII “O”)
- 0X55 (ASCII “U”)
- Value 1: hex digit in character format
  - character can be “0”, “1”, … “9”, “A”, …”F”
- Value 2: a second hex digit
- 0xA (ASCII ‘\n’)
- Value 3: the checksum for the rest of the packet

Lab 3: Packet Specification

- ASCII character representation:
  - "0" is 0x30
  - "1" is 0x31
  - "A" is 0x41
  - …

- Value 3: the checksum:

  0x4F + 0x55 + value1 + value2 +value3 + 0x0A

Lab 3: Packet Processing

- If none of the required elements match, then immediately drop the packet & start listening for more
- Checksum must match (otherwise, drop it)

Lab 3: Frame Processing

A single byte frame will consist of:

- A “0” start bit
- 8 data bits with the most significant bit first
- Two “0” stop bits

If there is not a valid start bit or the stop bits are missing:

- Drop the frame

Lab 3: Bit Encoding

- A bit will be 10 msec in width

Infrared Communication

- Directional (10-40 degrees)
- Local (0-4 m)
- Potentially very low power (relative to RF)

How do we generate a signal for the infrared receivers that we have?

Last Time

- Serial communication
- Lab 3: Infrared serial comm
  - Due April 14th
  - Get started this week!

Today

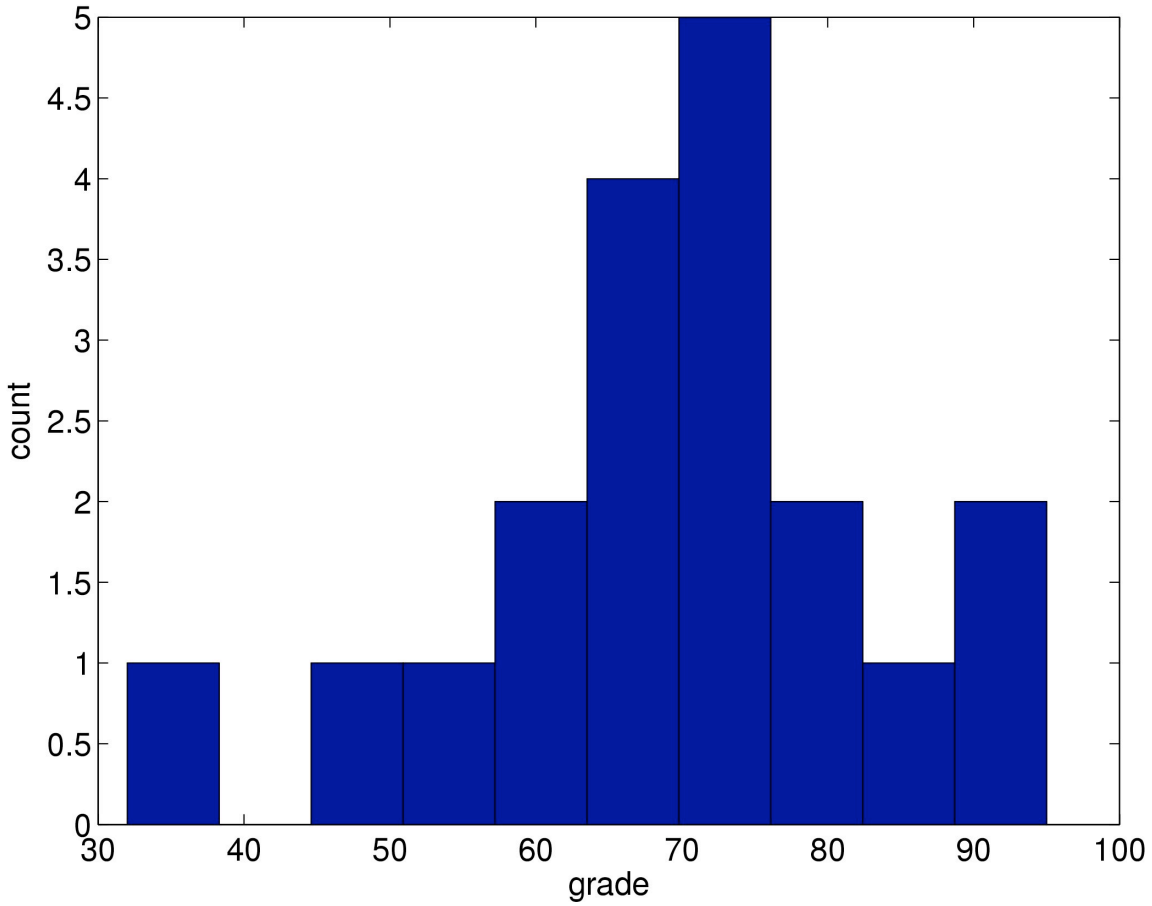
- Complete serial communication
- Architecture

Administrivia

- Midterms are graded
- Any grading questions must be addressed before the exams leave the classroom (or else they must be returned to me until we can talk)

Midterm Grades

- Max: 95
- Mean: 69
- Standard deviation: 14



Serial Communication

- Definition: transmission of information over time
  - In the extreme case, we transmit only one bit of information at a time
- In all but the best-controlled conditions, we have to assume that there will occasionally be transmission errors

Dealing with Errors

- We use a “hierarchy” of stages to interpret the serial data
- Each stage:
  - Represents a larger “chunk” of data
  - Provides its own mechanisms to ensure transmission or to detect errors
- What are these stages?

Serial Communication Stages

- A single bit
- A single byte (or “data frame”)
- A data packet

Serial Communication Stages

- A single bit
  - Processing depends heavily on the hardware used for transmission/reception
  - We transmit the value of the bit over some period of time. This allows the receiver to take multiple samples of the bit
- A single byte(or “data frame”)
- A data packet

Serial Communication Stages

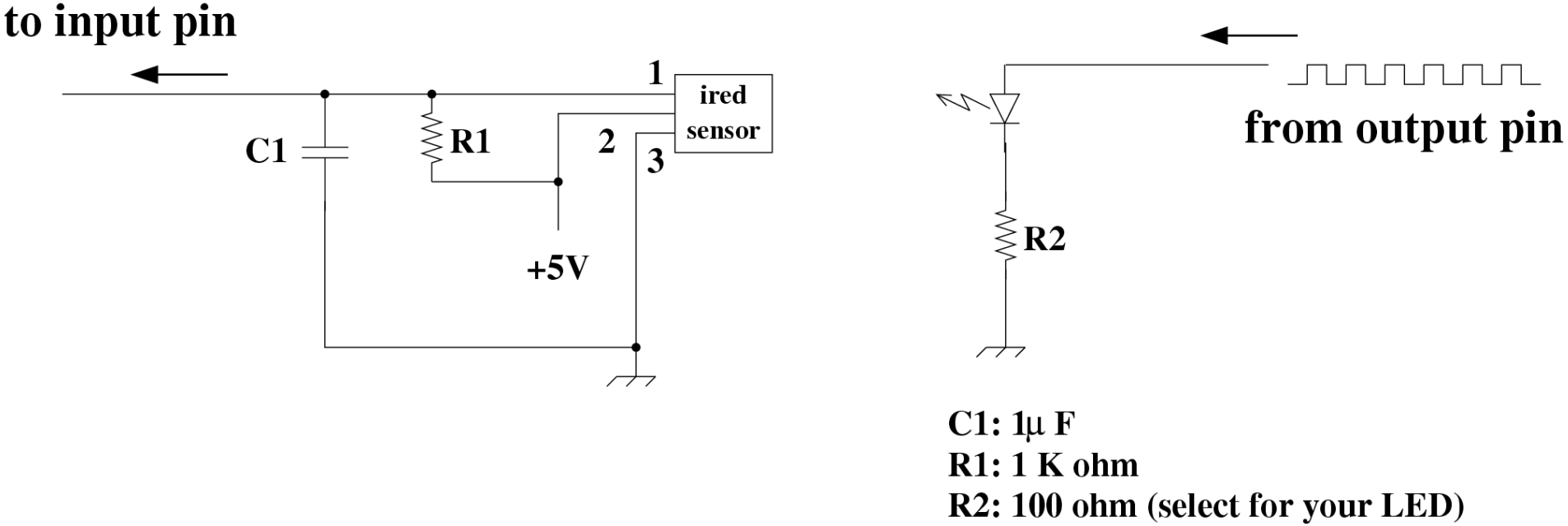
- A single bit
- A single byte (or “data frame”)
  - Start bit:
    - Indicates when a byte is coming
    - Allows the receiver to synchronize its clock with the transmitter
  - Stop bits
    - Allow the receiver to check that the byte is “framed” correctly
  - Parity bit(s)
    - Allow for a simple check on the correctness of the byte
- A data packet

Serial Communication Stages

- A single bit
- A single byte (or “data frame”)
- A data packet
  - A collection of bytes that is specific to the application (problem) that you are solving
  - Often contains a number of bytes with constant values
  - A **checksum** byte allows us to check the entire content of the packet

Infrared Serial Implementation

- Recall that our Ired receivers require a 40 Khz carrier
- So – a single bit would be indicated using a sequence of LED pulses



Encoding a Single Bit

- With our Ired sensors, signal reception is indicated as a “low” on the sensor output
- The output of the sensor is significantly filtered
- When reading “low”, the line will only be low about 38% of the time (sensor property)
- “high” will be encoded as high 100% of the time (ideally)

Encoding a Single Bit

How can we be more sure about the true bit value?

- Sample the state of the sensor many times over the course of a single bit
- Above some percentage of “lows” we will interpret the bit as a “0”
- Otherwise, it is a “1”

Bit Encoding

Let's make the following (somewhat arbitrary) choice:

- Encode a bit value of “0” with the transmission of an infrared signal
- Encode a value of “1” with no transmission

Infrared Serial Implementation

If our goal is to minimize power usage, how should we structure our data frame?

Encoding a Byte

For lab 3, a data frame is encoded as follows:

- 1 start bit of value “0” (why not “1”?)
- 8 bits of data
- 2 “0” stop bits

Could also include a parity bit (but not for this lab)

Reading a Valid Byte

```
int read_next_byte() {
  wait_for_start_bit();
  // Read the 8 bits
  for(i = 0, out = 0; i < 8; ++i) {
     out = out << 1 + read_bit();
  }
  stop0 = read_bit();
  stop1 = read_bit();
  if(stop0 == 1 || stop1 == 1) return(-1);
  return(out);
}
```

Reading a Valid Byte

`read_next_byte()`

- Returns an integer
- An int is a 16-bit, signed number
- Return values 0-255 are valid byte values
- Return value of -1 means that an error has occurred

Reading a Valid Byte

`wait_for_start_bit()`

- Returns at the end of the start bit

`read_bit()`

- Multiple sensor samples
- Returns at the end of the bit

How do we handle the timing?

Reading a Valid Byte

How do we handle the timing?

- Sample at regular intervals (remember that a bit will be 10 msec in width)
- Detecting the start bit:
  - Wait for the line to go low
  - Sample at regular intervals
  - If the line stays low on average, timeout and return
  - Else: go back to waiting

Packet Specification (Lab 3)

A valid packet will be composed of a specific sequence of bytes:

- 0x4F (ASCII “O”)
- 0X55 (ASCII “U”)
- Value 1: hex digit in character format
  - character can be “0”, “1”, … “9”, “A”, …”F”
- Value 2: a second hex digit
- 0xA (ASCII ‘\n’)
- Value 3: the checksum for the rest of the packet

Receiving a Packet

```
int read_packet() {
   uint8_t csum;
   int val1, val2;
   if(read_byte() != 'O') return(-1);
   if(read_byte() != 'U') return(-1);
   if((val1 = read_byte()) == -1) return(-1);
   if((val2 = read_byte()) == -1) return(-1);
   if(read_byte() != '\n') return(-1);
   csum = read_byte();
   if(csum != ('0' + 'U' + val1 + val2 +          '\n')
   return(-1);
   return(ascii2int(val1) << 4 +
             ascii2int(val2));   // Valid packet!
}
```

Hints

- Watch out for interference from other devices
- The florescent lights (and the sun) will produce spurious sensor readings

Components of a Computer

- Microprocessor
- Memory
  - Both RAM and ROM
- Input/Output devices
- Clock

Last Time

- Serial communication
- Infrared transmission/reception
- Lab 3
  - Get started this week!
  - See me or Mark for “the demo”

Today

- A little more C code
  - (and lab hints)
- More computer architecture

“Funky” C Syntax

What does this code do?

```
if((val1 = read_byte()) == -1)
  return(-1);
```

“Funky” C Syntax

```
if((val1 = read_byte()) == -1)
  return(-1);
```

This is equivalent to:

```
val1 = read_byte();
if(val1 == -1) return(-1);
```

Receiving a Packet (An Update)

```
int read_packet() {
   uint8_t csum;
   int val1, val2;
   if(read_byte() != 'O') return(-1);
   if(read_byte() != 'U') return(-1);
   if((val1 = read_byte()) == -1) return(-1);
   if((val2 = read_byte()) == -1) return(-1);
   if(read_byte() != '\n') return(-1);
   csum = read_byte();
   if(csum != 0x00FF &
       ('0' + 'U' + val1 + val2 +'\n')) return(-1);
   return(ascii2int(val1) << 4 +
             ascii2int(val2));   // Valid packet!
}
```

How Would We Implement ascii2int()?

How Would We Implement ascii2int()?

```
int ascii2int(int val) {
  char c;

  c = (char) val;
  if(c >= '0' & c <= '9')
        return(c - '0');
  else   //  c in {'A' … 'F'}
        return(10 + c - 'A');

}
```

Components of a Microprocessor

- Registers (fast-access memory)
  - General purpose: used for data storage
  - Special purpose: used to control the behavior of the microprocessor and/or the devices connected to it
- Instruction decoder
  - Instructions are the primitive “actions” that the microprocessor can perform
  - Load/store to/from memory, AND, ADD, jump…

Components of a Microprocessor

- Arithmetic Logical Unit (ALU)
- Memory control logic
- Timers
  - Including timing mechanisms for instruction fetch and execution
- Interrupt processor

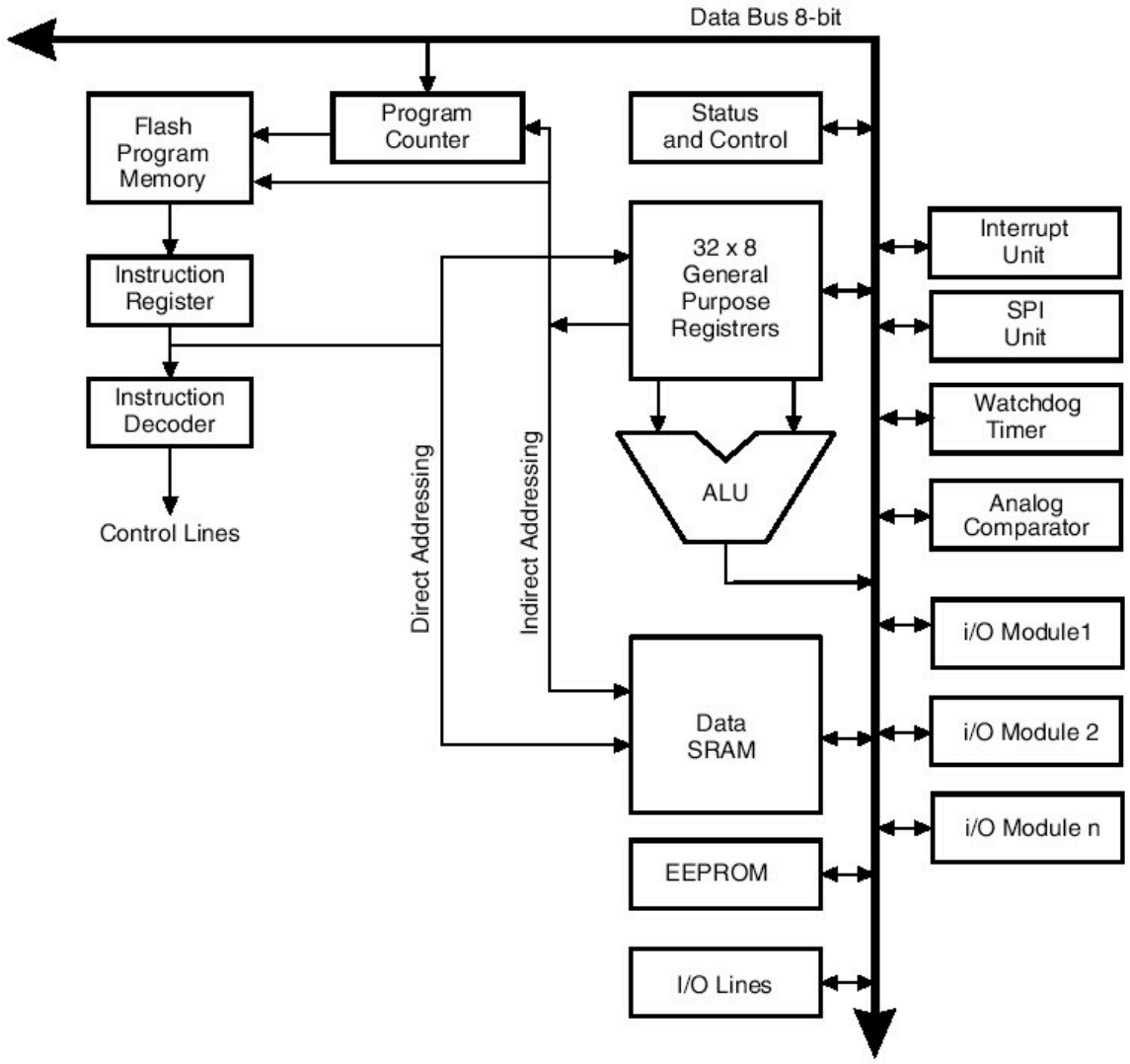
Common Special-Purpose Registers

- Program counter (PC): address of memory from which the next instruction will be fetched
- Status register (SR): stores basic state of the processor
- Instruction register: stores the recently fetched instruction

Common Special-Purpose Registers

- Stack pointer (SP): address in memory of the top of “the stack”
  - On a subroutine call, will store the return address and relevant data
  - When the subroutine “returns” (ends), the PC address is read back from the stack

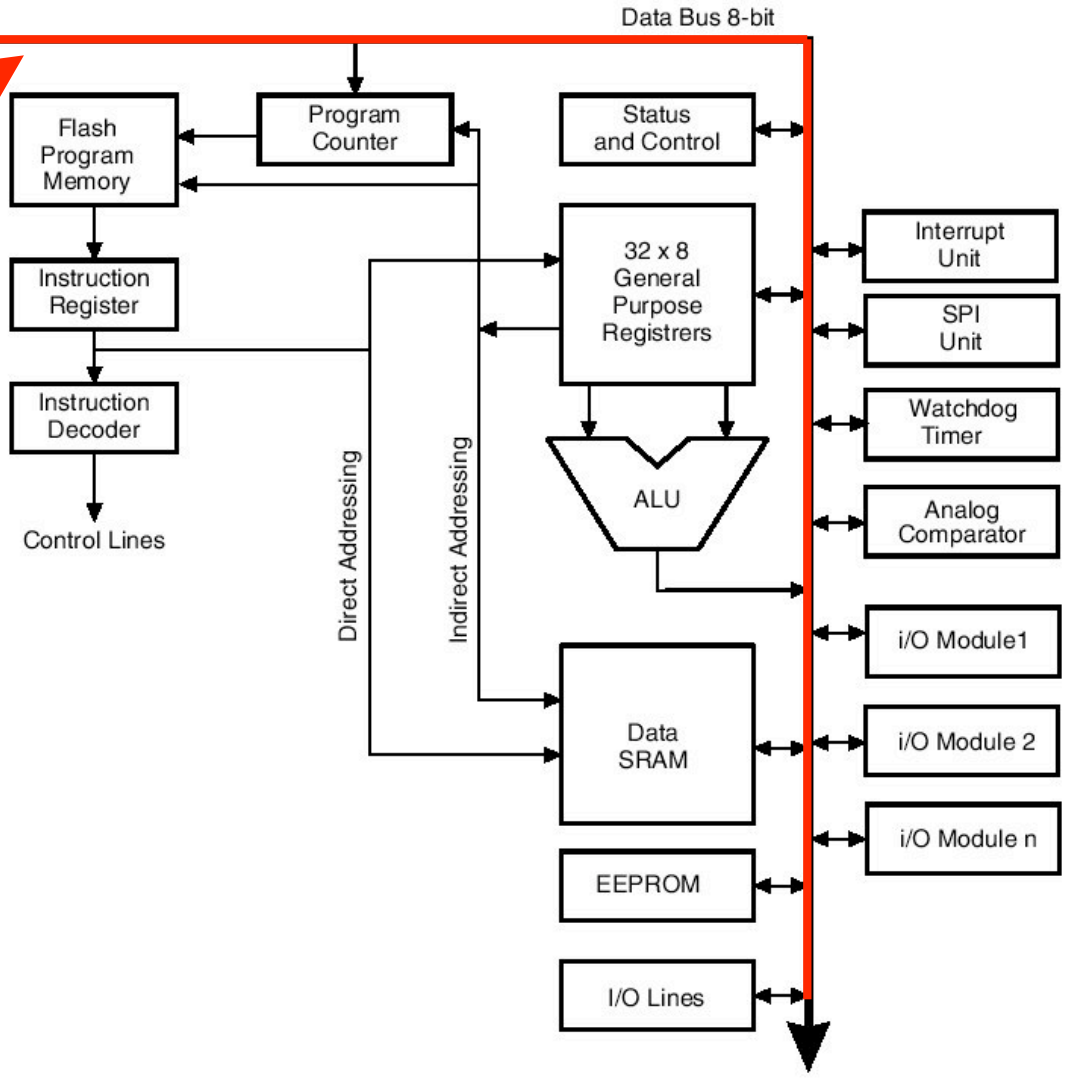
An Example: the Atmel Mega8



Atmel Mega8

8-bit data bus

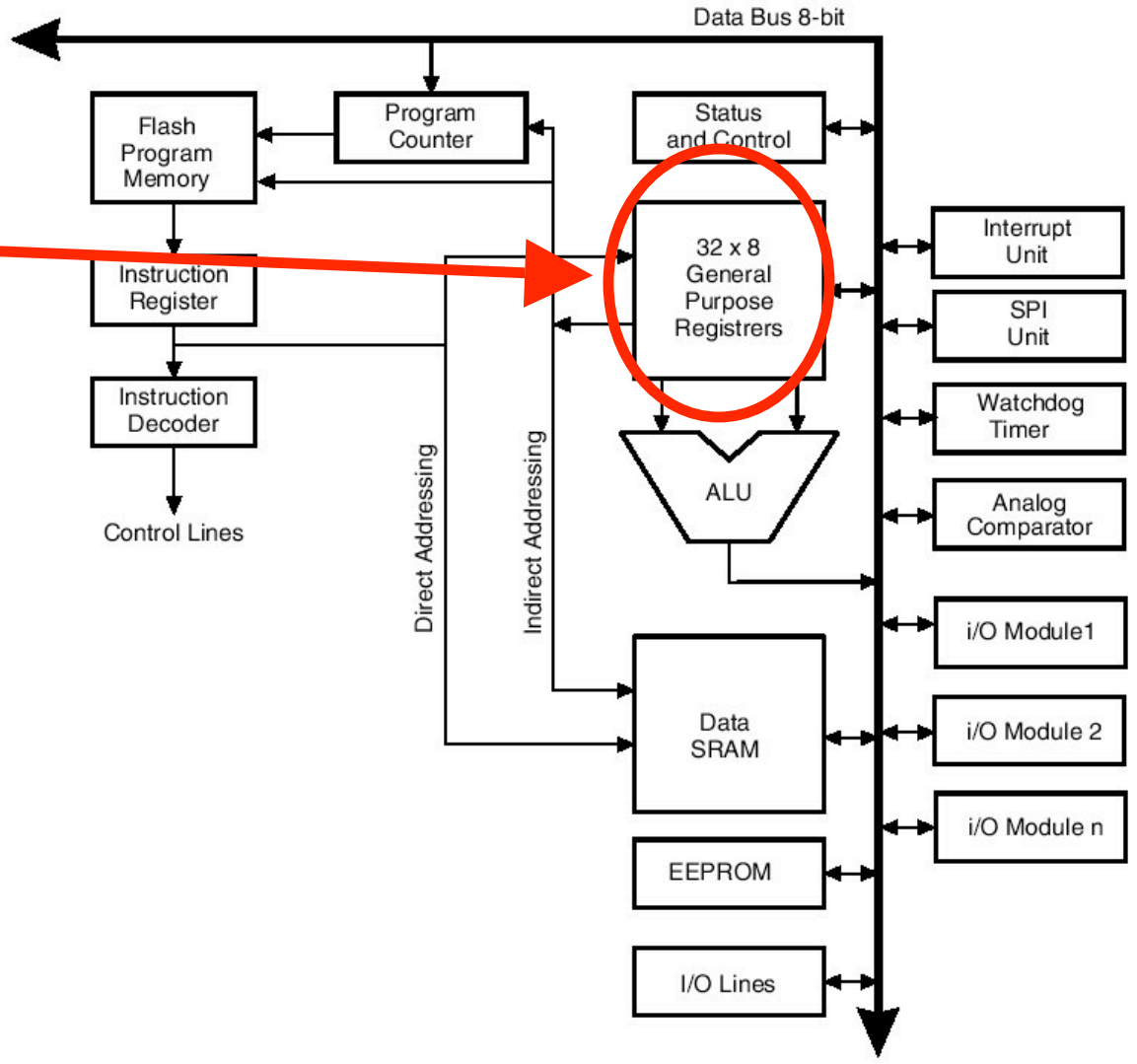
- Primary mechanism for data exchange



Atmel Mega8

32 general purpose registers

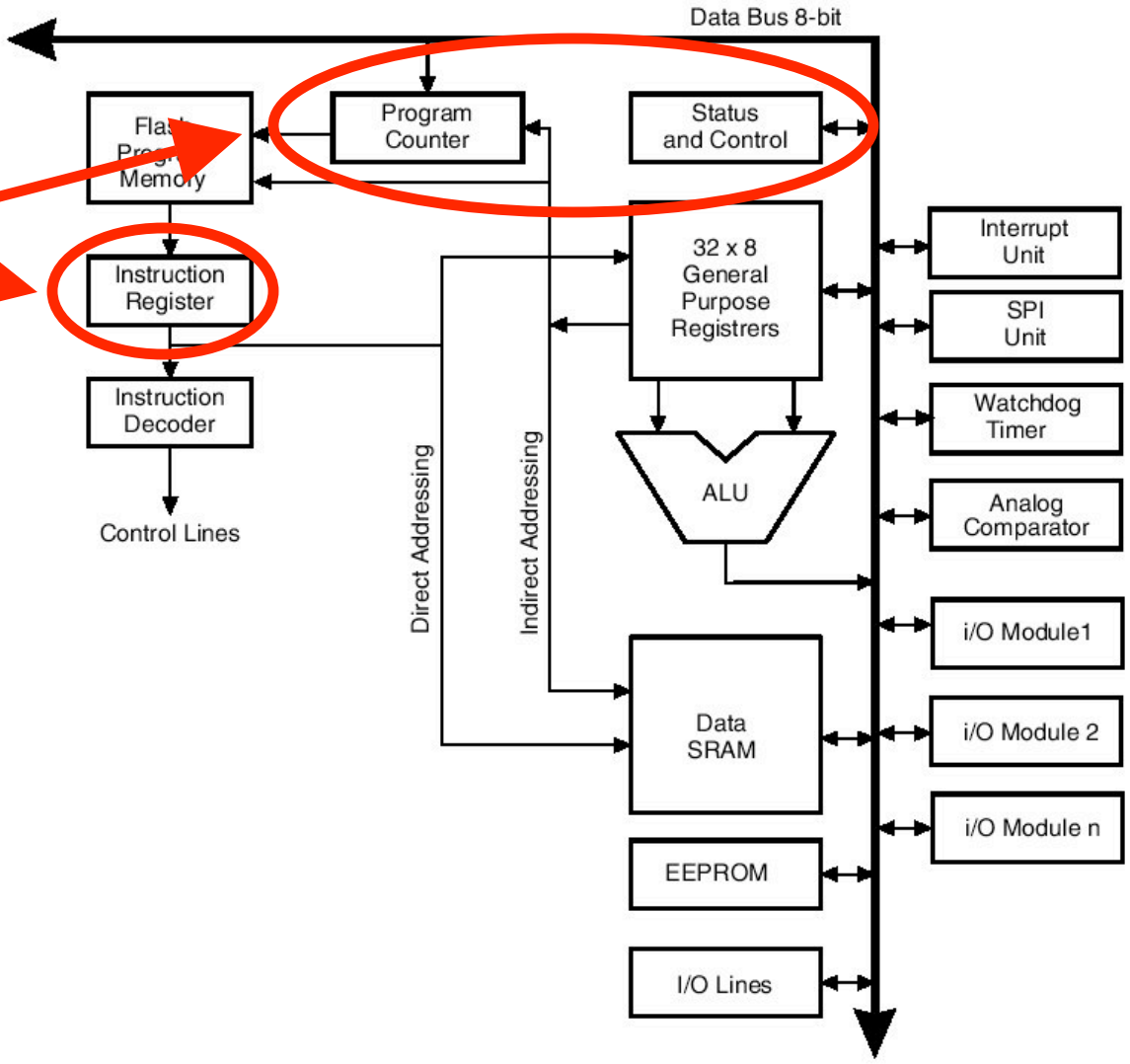
- 8 bits wide
- 3 pairs of registers can be combined to give us 16 bit registers



Atmel Mega8

Special purpose registers

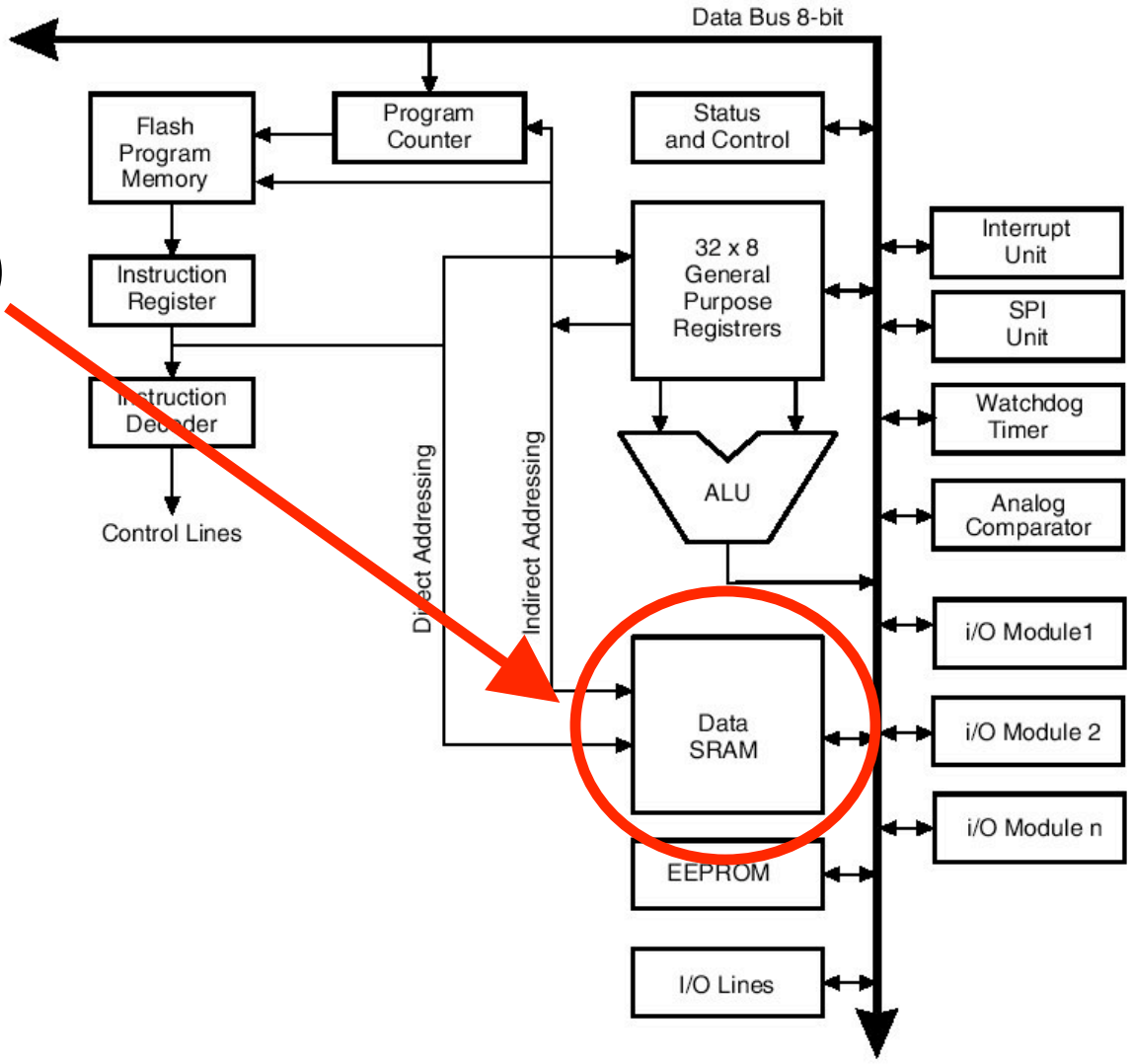
- Control of the internals of the processor



Atmel Mega8

Random Access Memory (RAM)

- 1 KByte in size

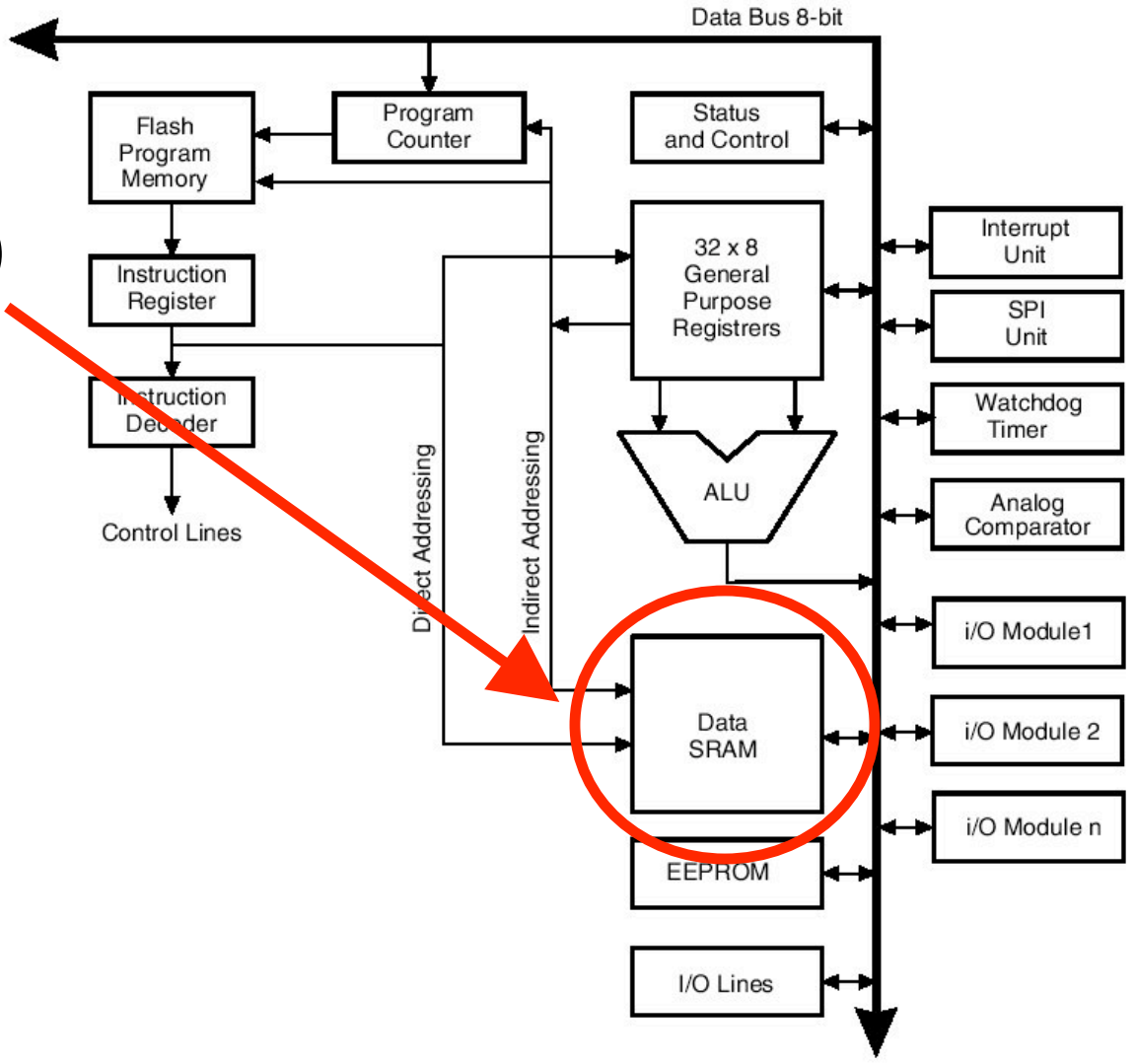


Atmel Mega8

Random Access Memory (RAM)

- 1 KByte in size

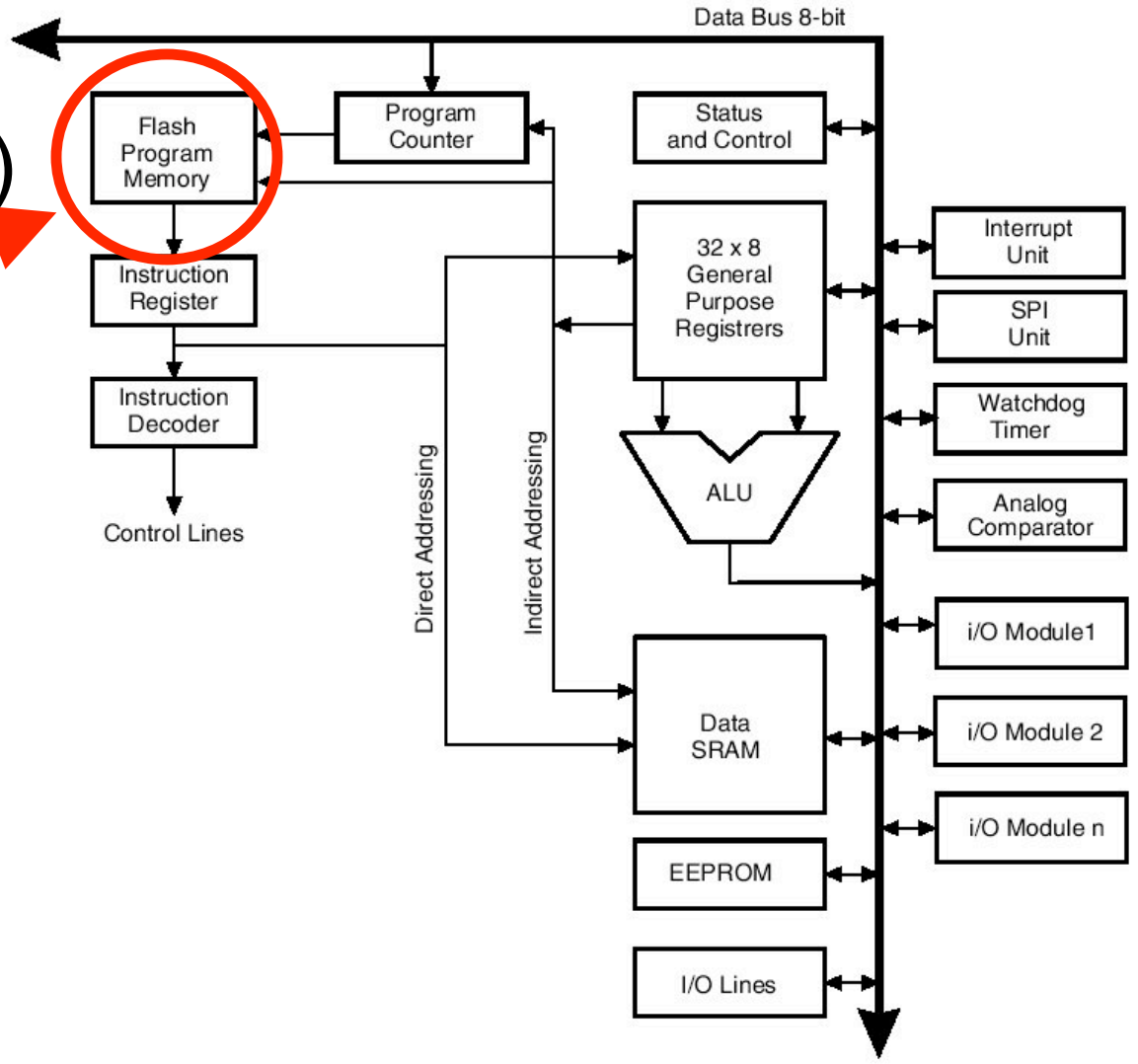
Note: in high-end processors, RAM is a separate component



Atmel Mega8

Flash (EEPROM)

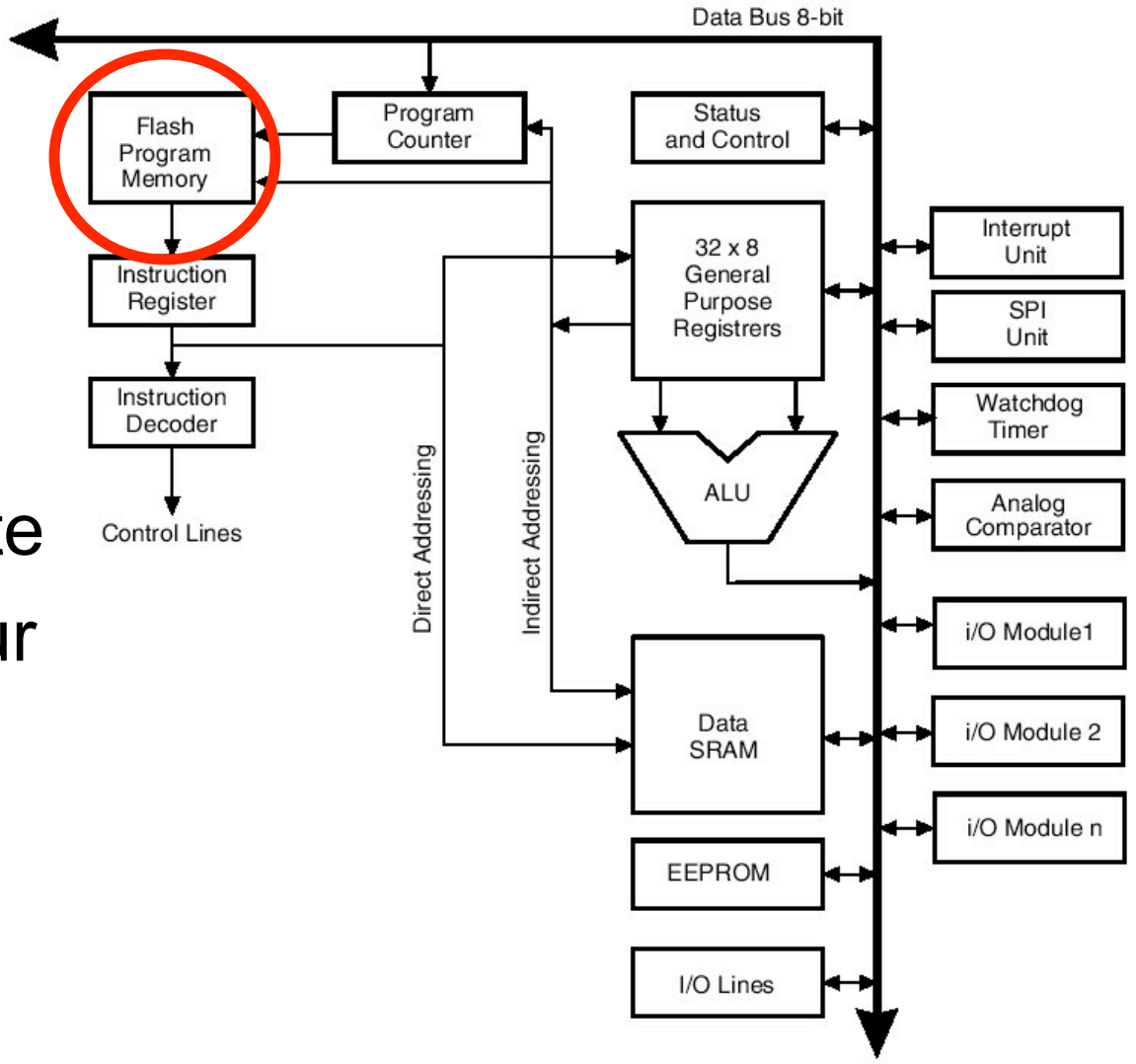
- Program storage
- 8 KByte in size



Atmel Mega8

Flash (EEPROM)

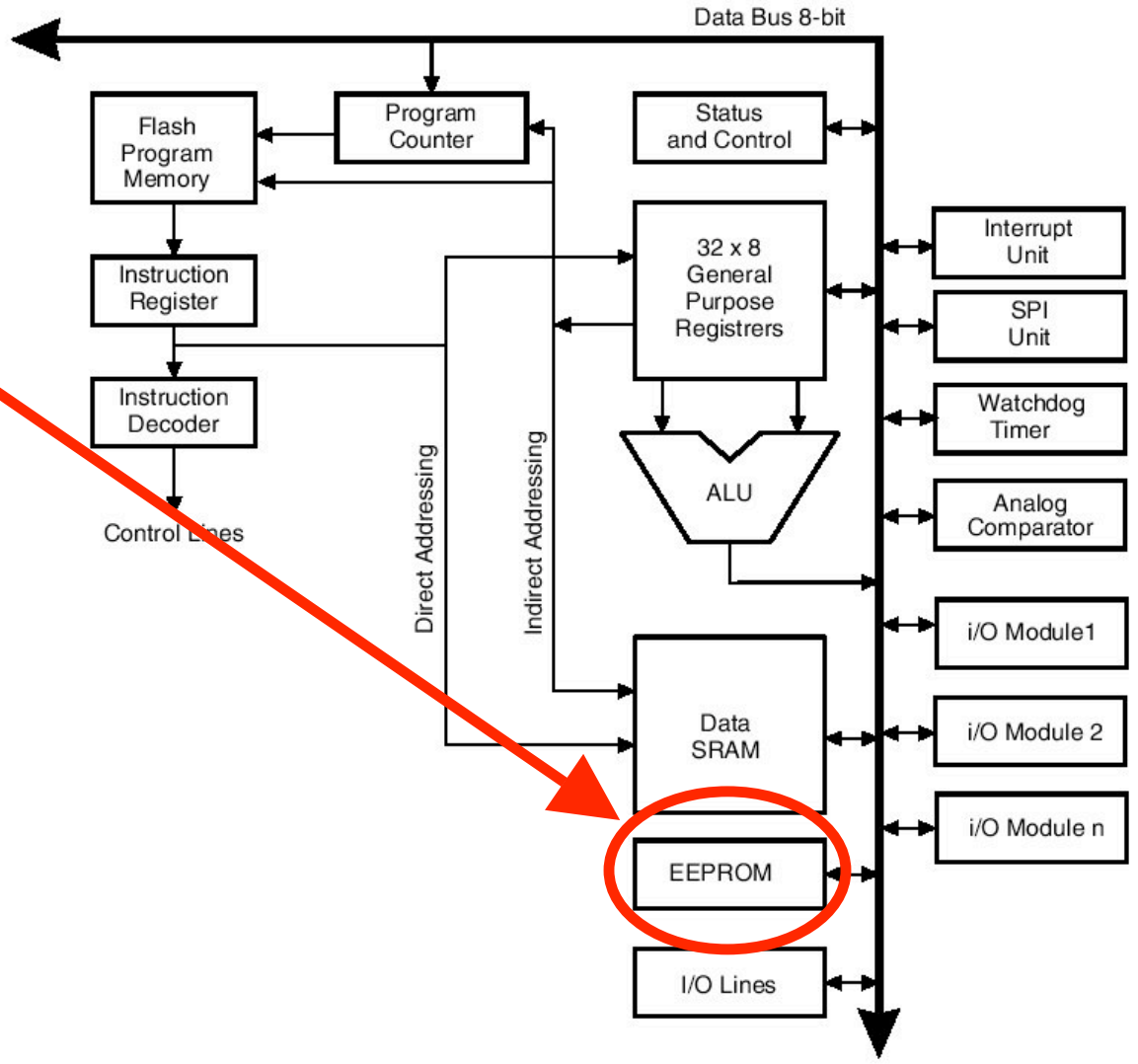
- In this and many microcontrollers, program and data storage is separate
- Not the case in our general purpose computers



Atmel Mega8

EEPROM

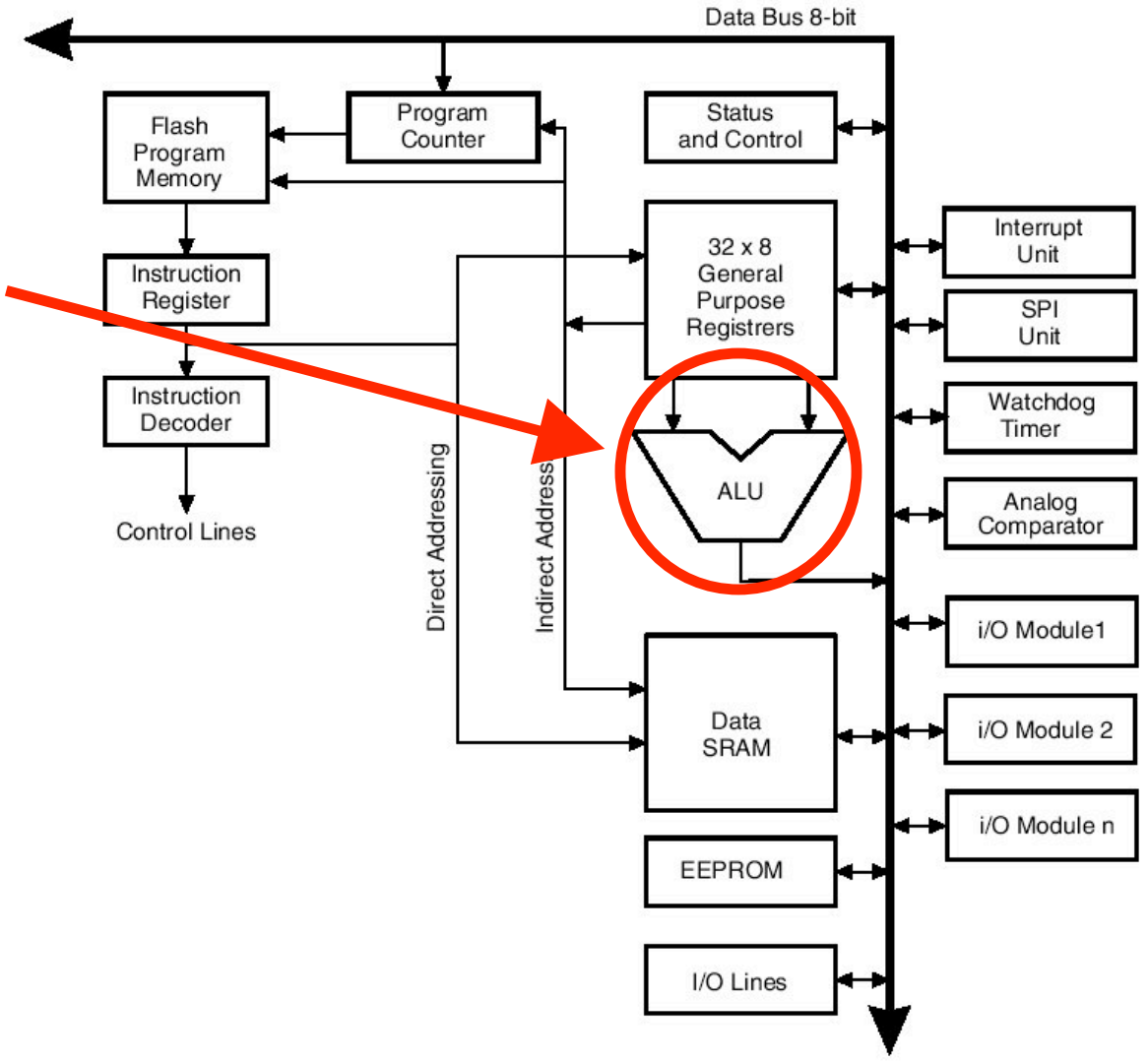
- Permanent data storage



Atmel Mega8

Arithmetic Logical Unit

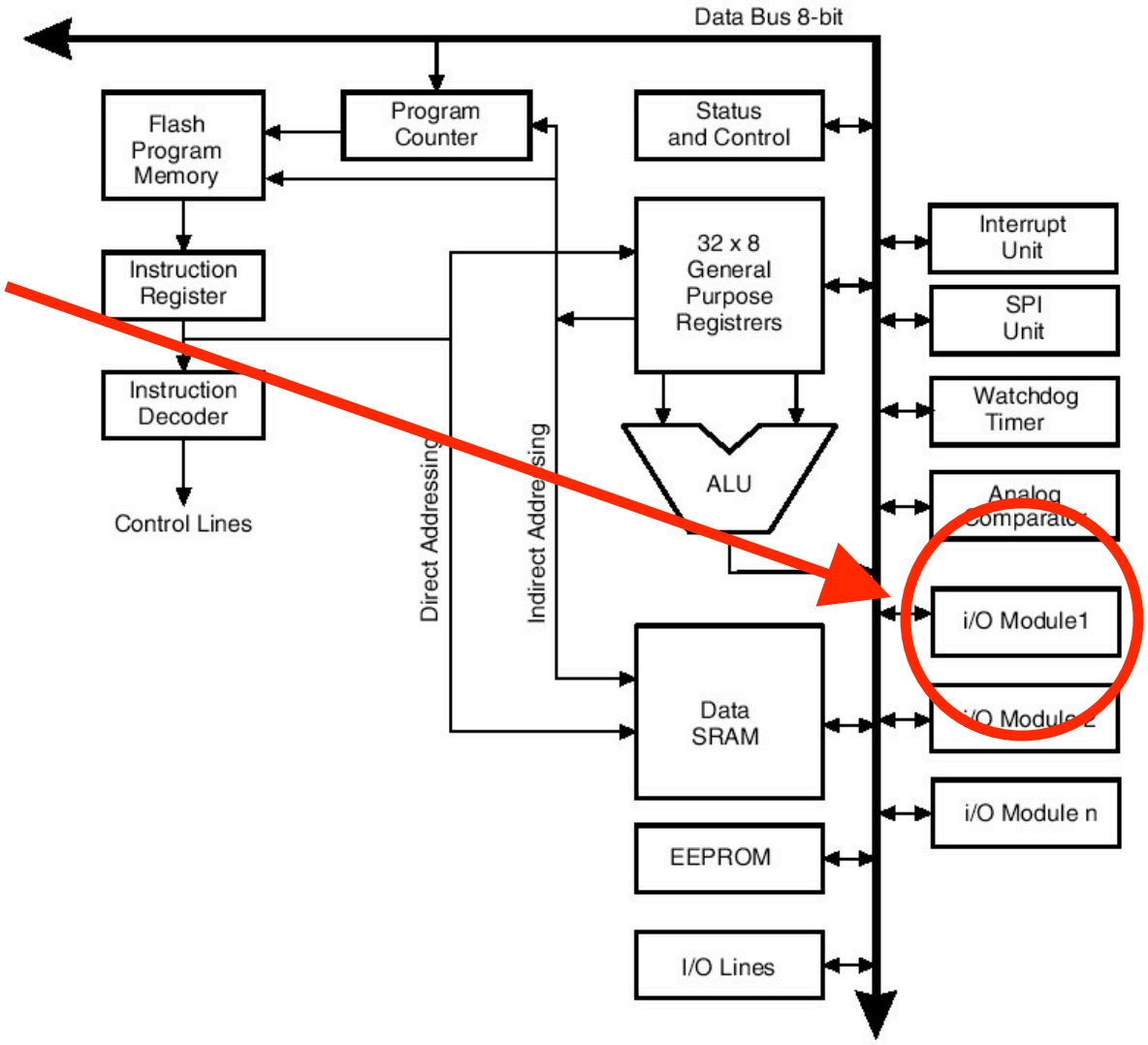
- Data inputs from registers
- Control inputs not shown (derived from instruction decoder)



Atmel Mega8

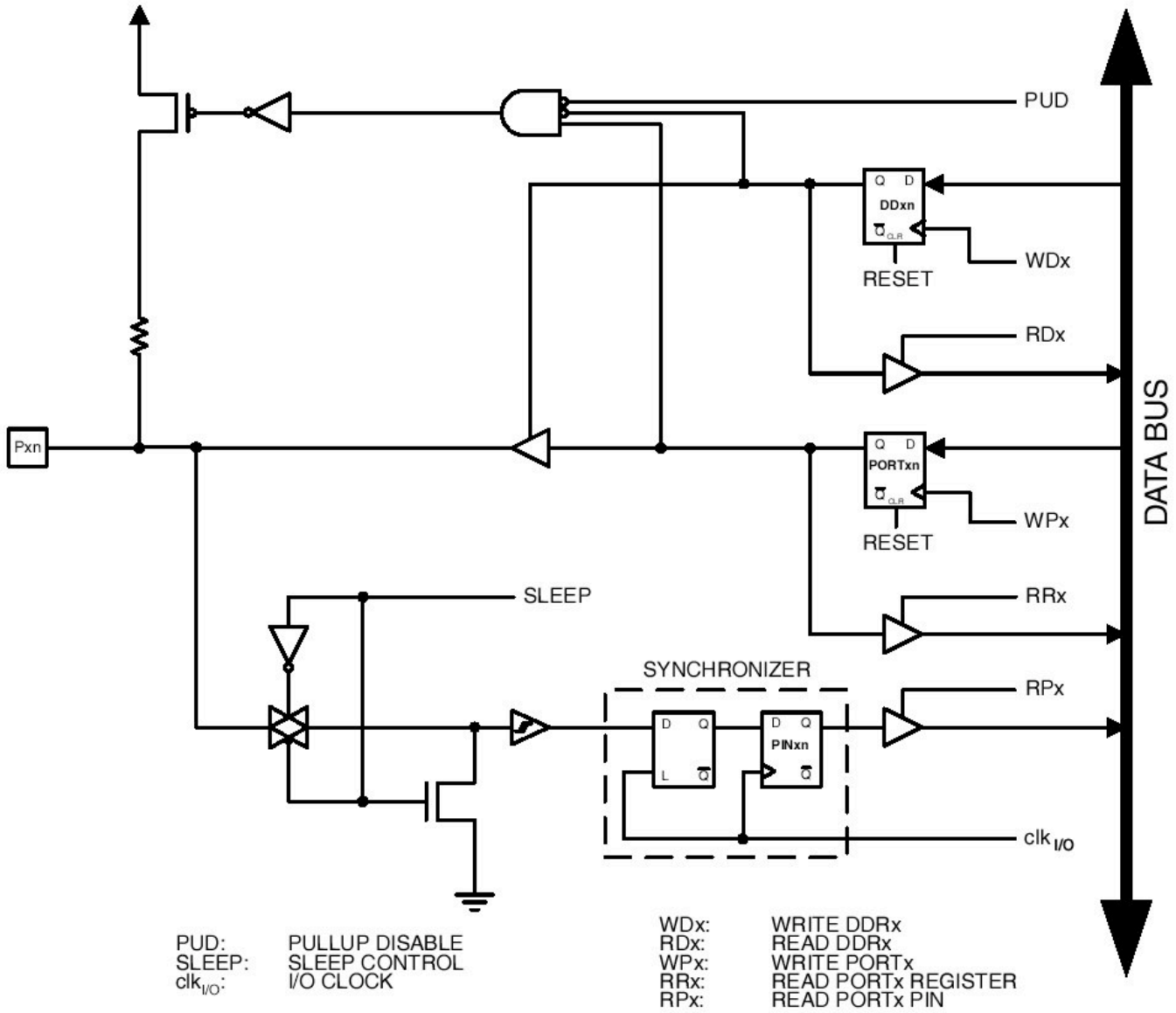
PORTB hardware

- Configurable as inputs or outputs



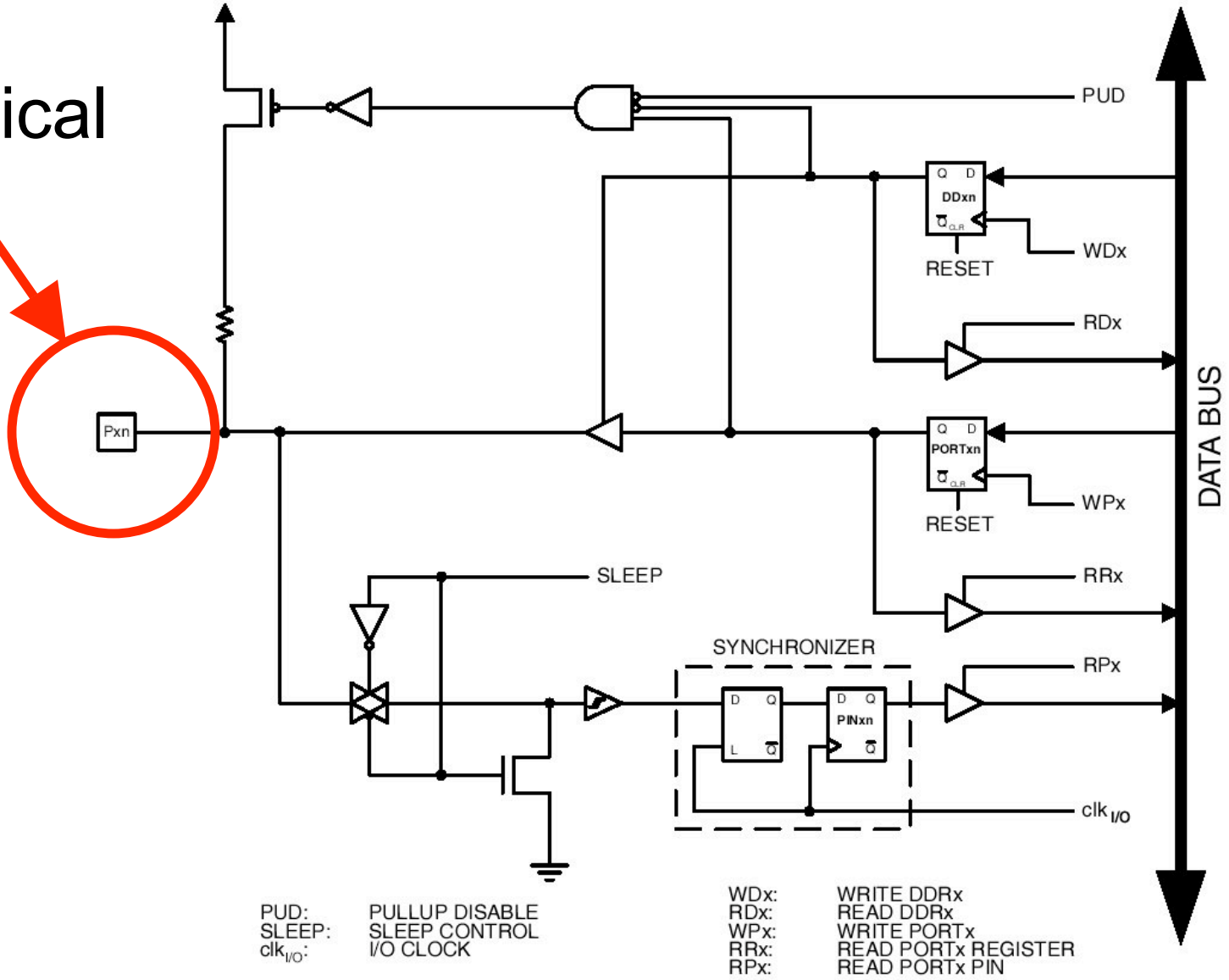
I/O Pin Implementation

Single bit of PORT B



I/O Pin Implementation

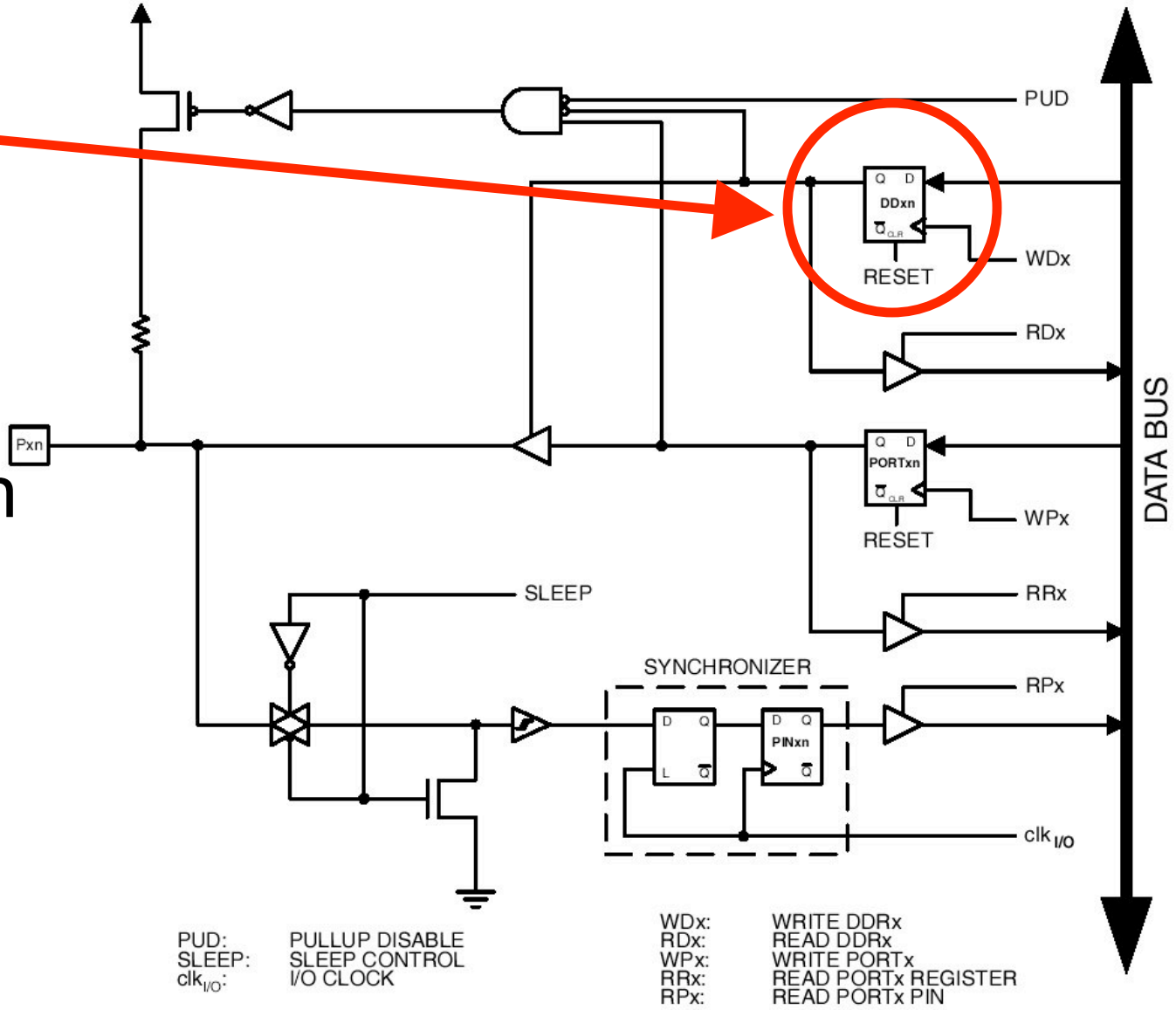
The physical pin



I/O Pin Implementation

DDRB

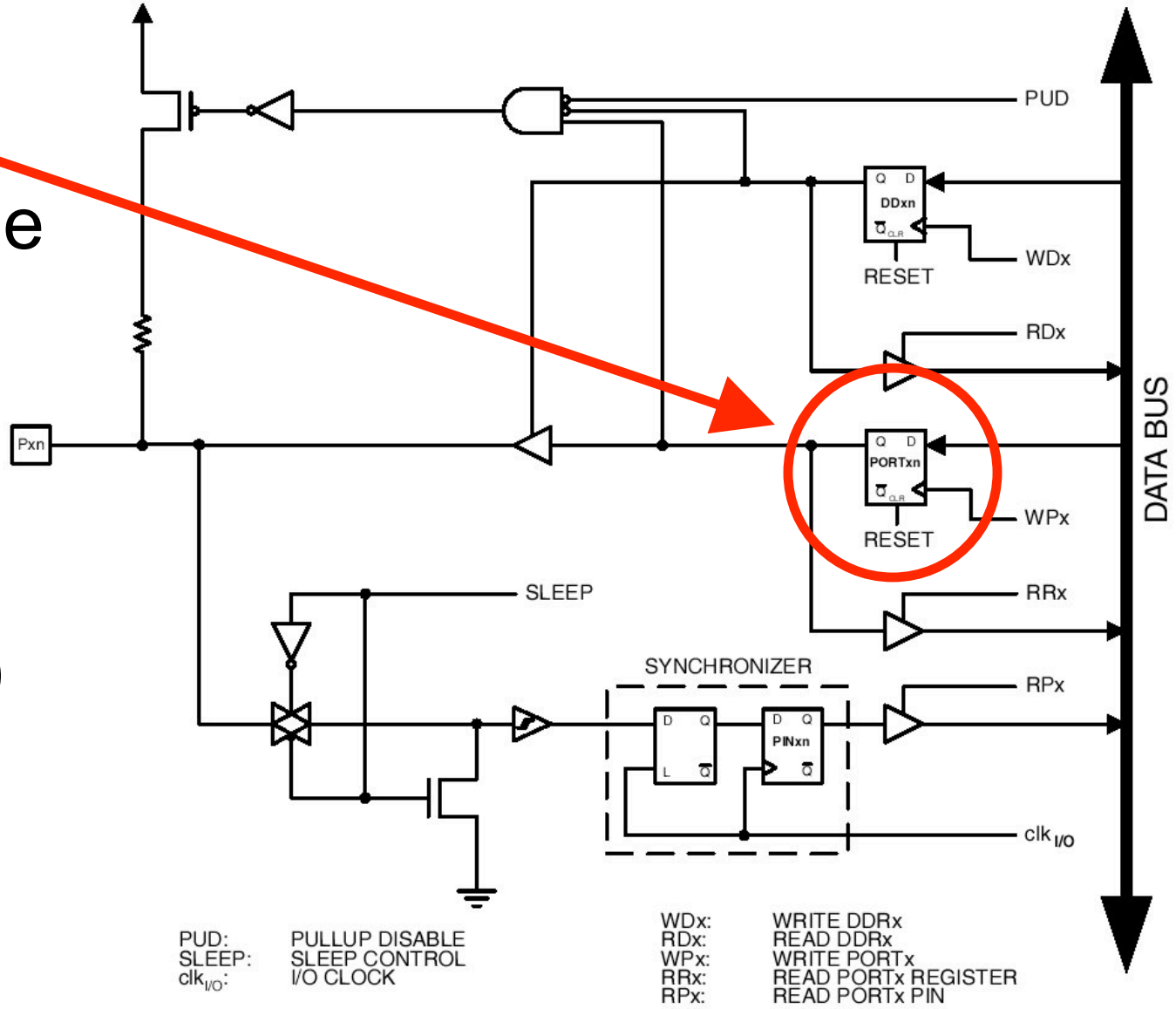
- Defines whether this is an input or an output



I/O Pin Implementation

PORTB

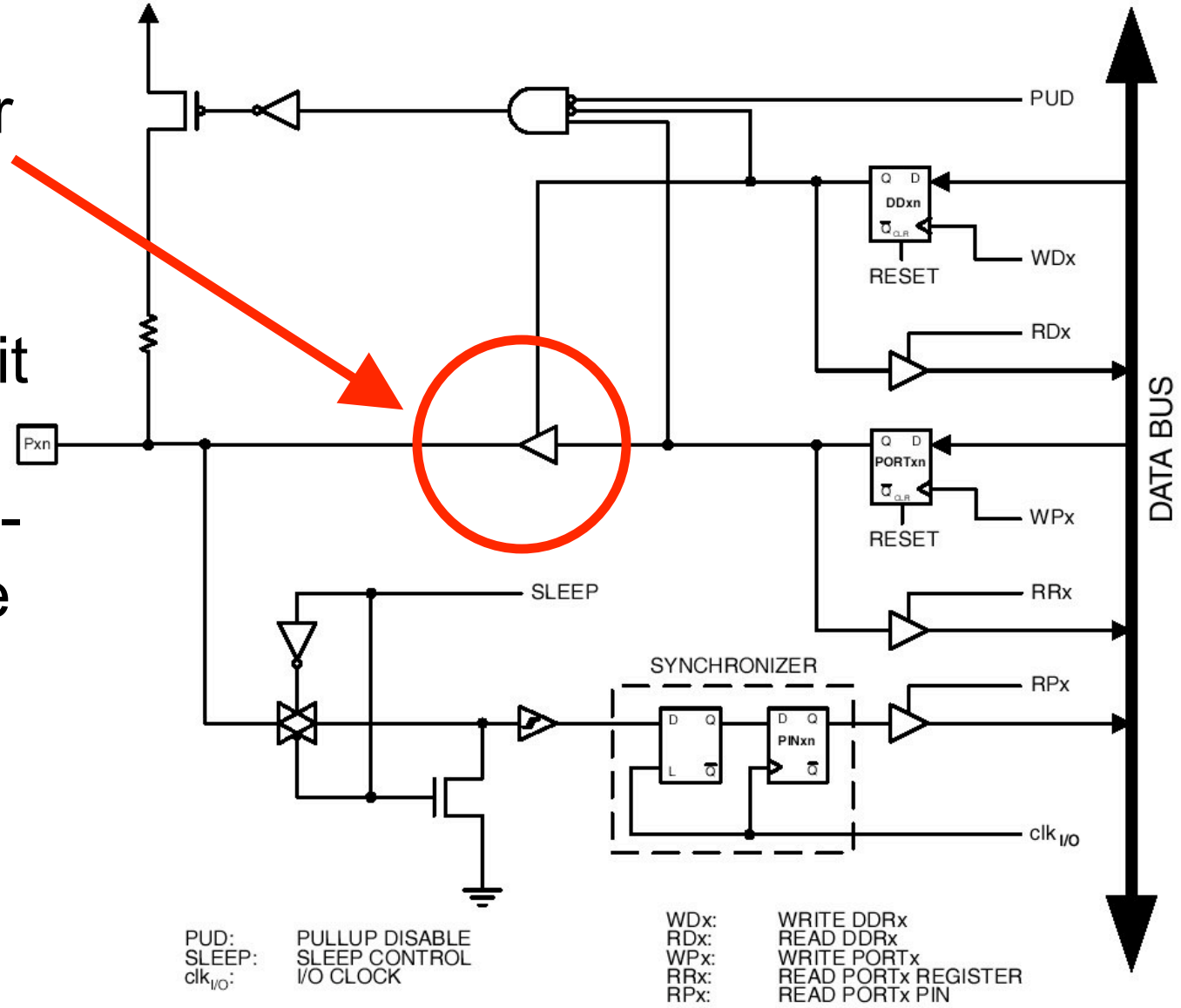
- Defines the value that is written out to the pin (if it is an output)



I/O Pin Implementation

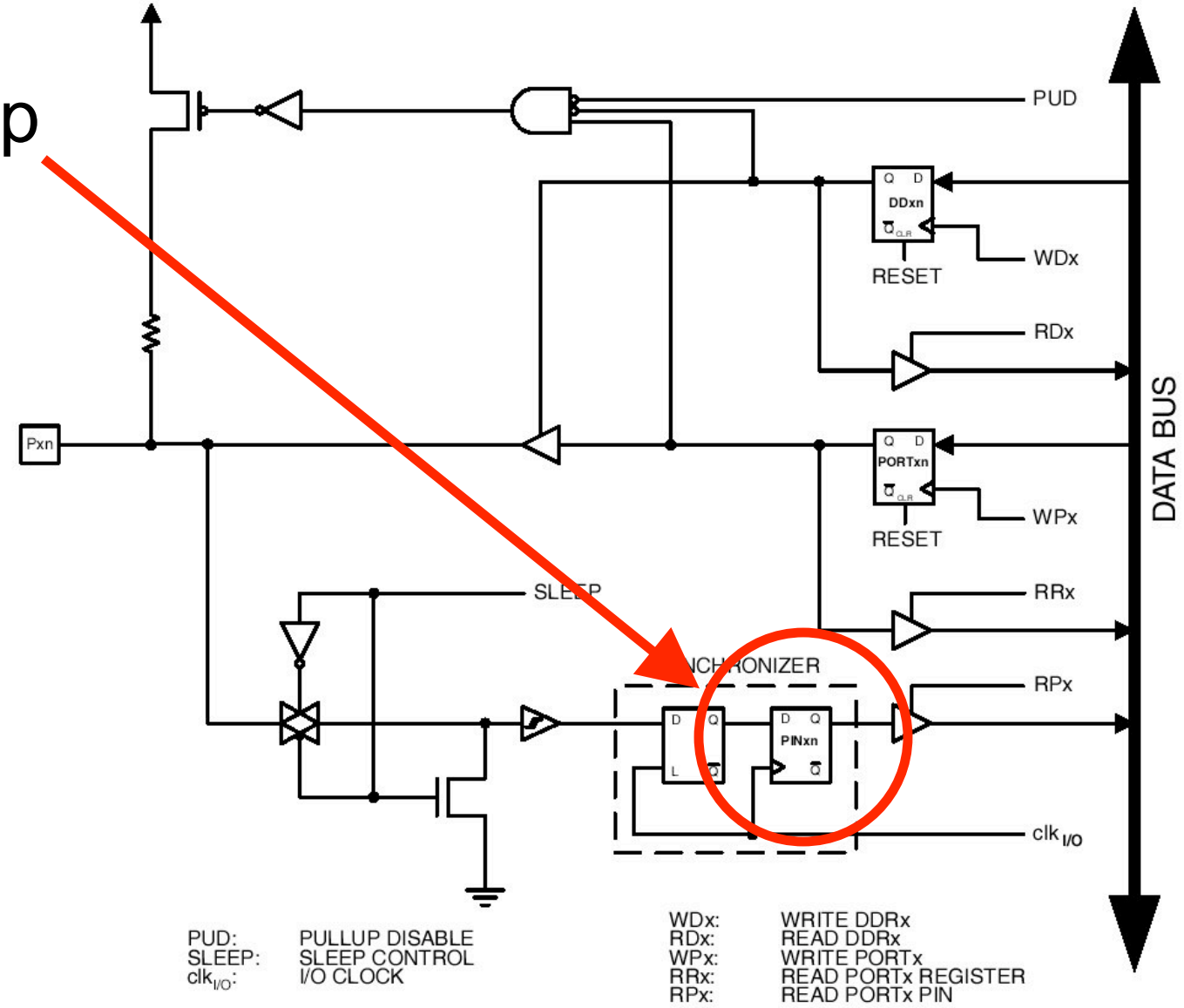
Tristate buffer

- When this pin is an output pin, it allows the PORTB flip-flop to drive the pin



I/O Pin Implementation

Input flip-flop



Last Time

- Serial processing
- Architecture of the mega 8

Today

- Midterm discussion
- Machine-level instructions and decoding

- Reading for this week and next: ESP Chapters 3 & 4

Administrivia

- Lab 3 due Thursday
  - Demonstrations outside of class
  - See the Atmel FAQ on blackboard

Machine-Level Programs

Machine-level programs are stored as sequences of machine instructions

- Stored in program memory
- Execution is generally sequential (instructions are executed in order)
- But – with occasional “jumps” to other locations in memory

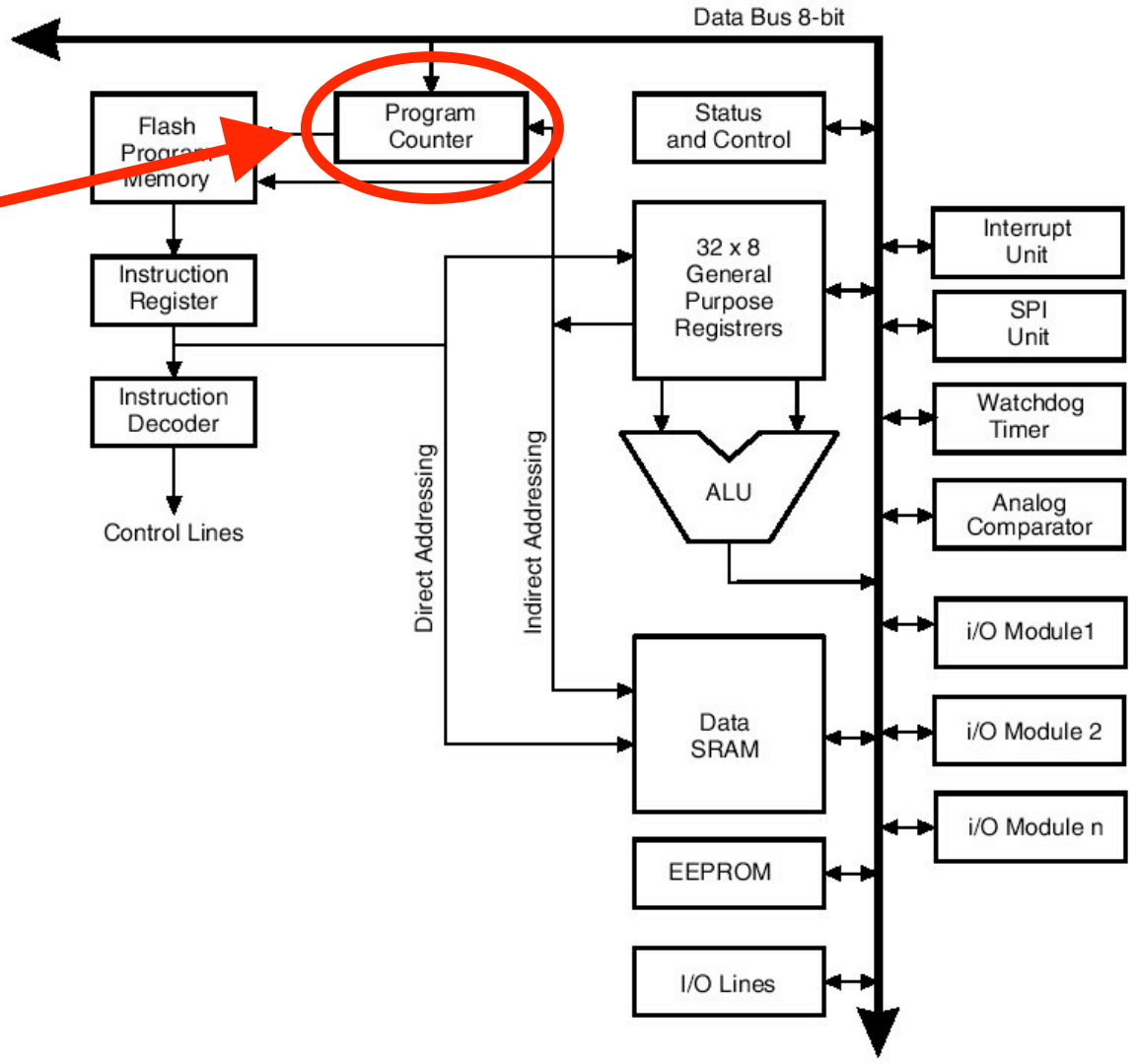
Types of Instructions

- Memory operations: transfer data values between memory and the internal registers
- Mathematical operations: ADD, SUBTRACT, MULT, AND, etc.
- Tests: value == 0, value > 0, etc.
- Program flow: jump to a new location, jump conditionally (e.g., if the last test was true)

Atmel Mega8: Decoding Instructions

Program counter

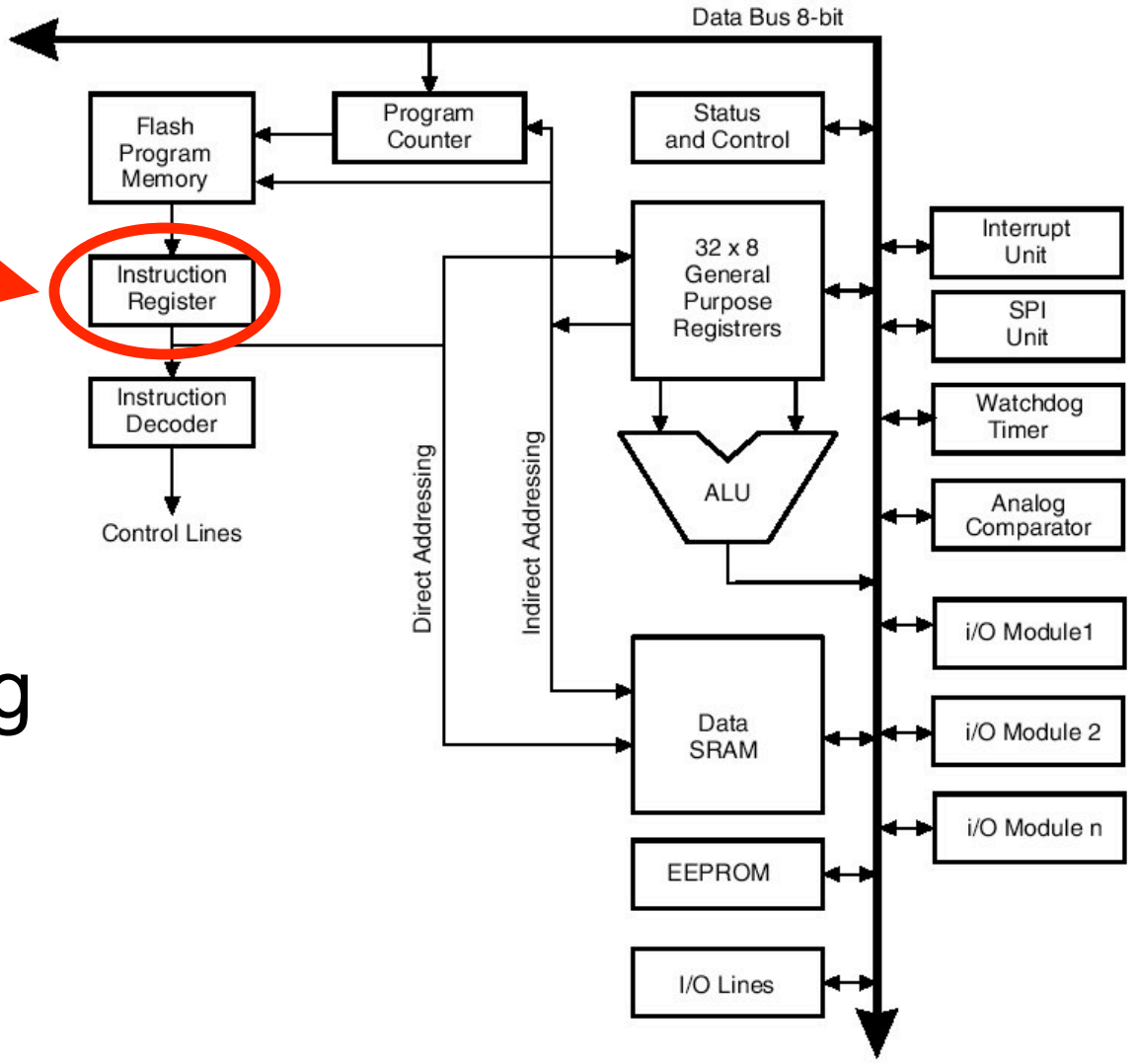
- Address of currently executing instruction



Atmel Mega8: Decoding Instructions

Instruction register

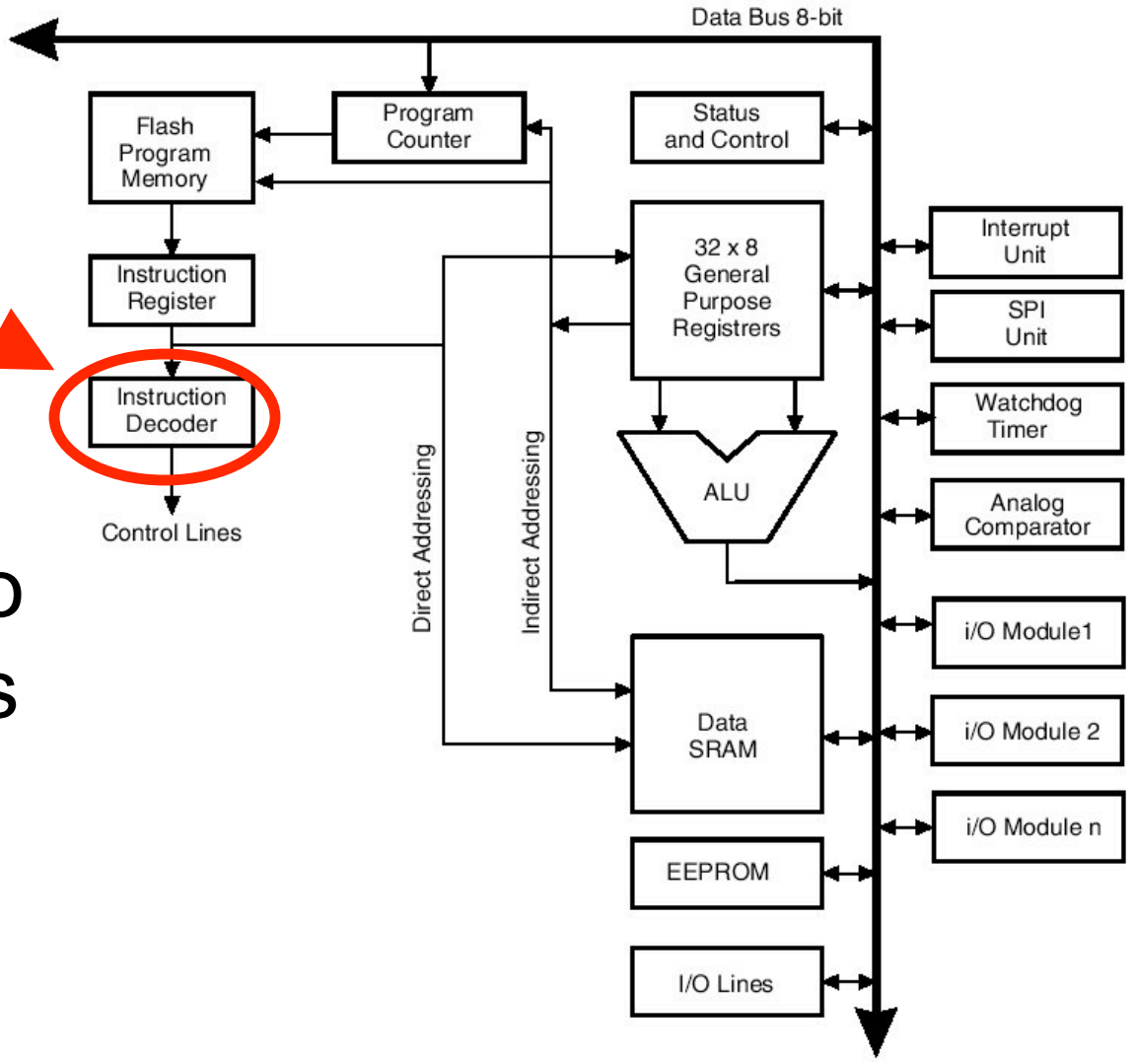
- Stores the machine-level instruction currently being executed



Atmel Mega8

Instruction decoder

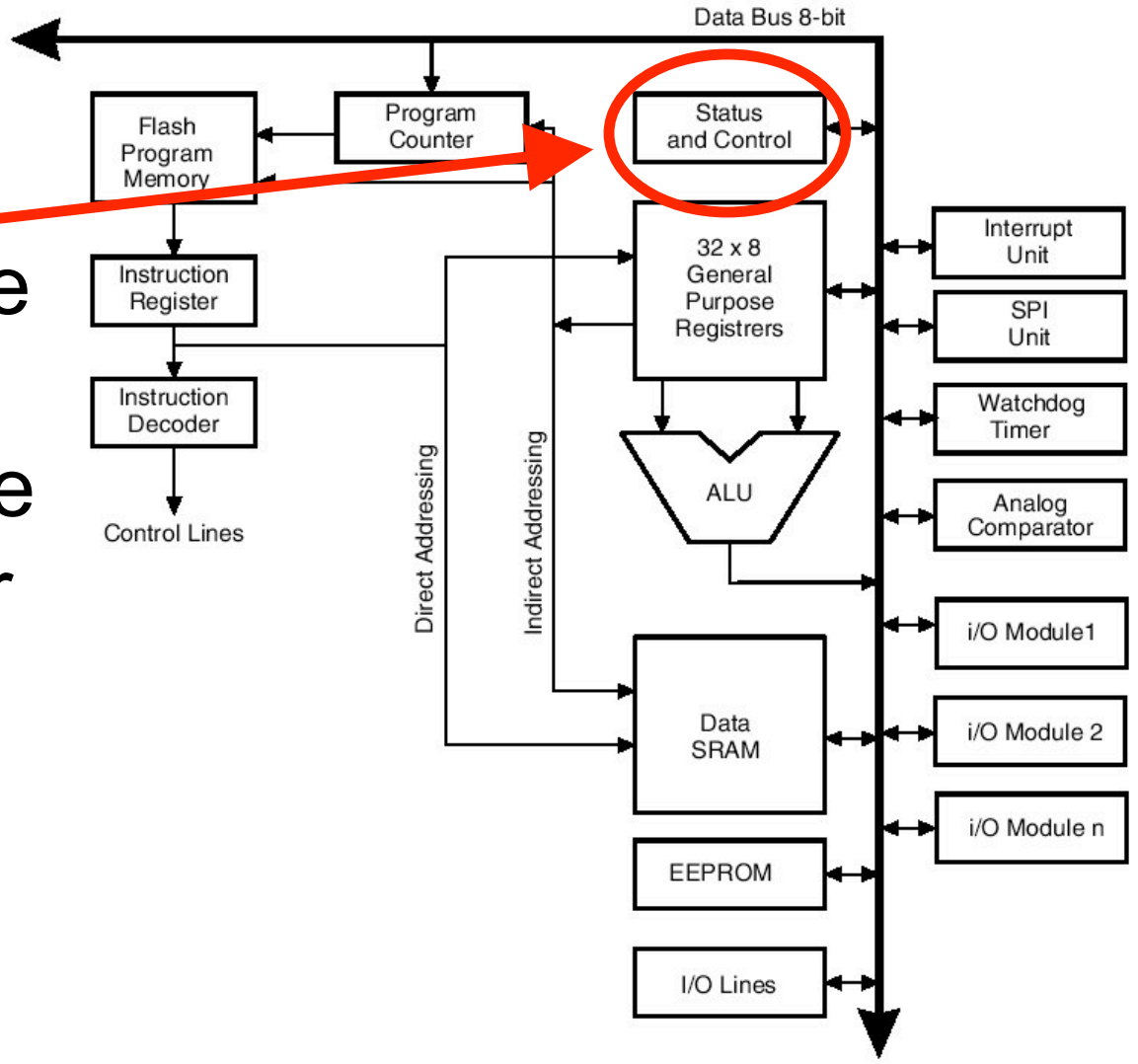
- Translates current instruction into control signals for the rest of the processor



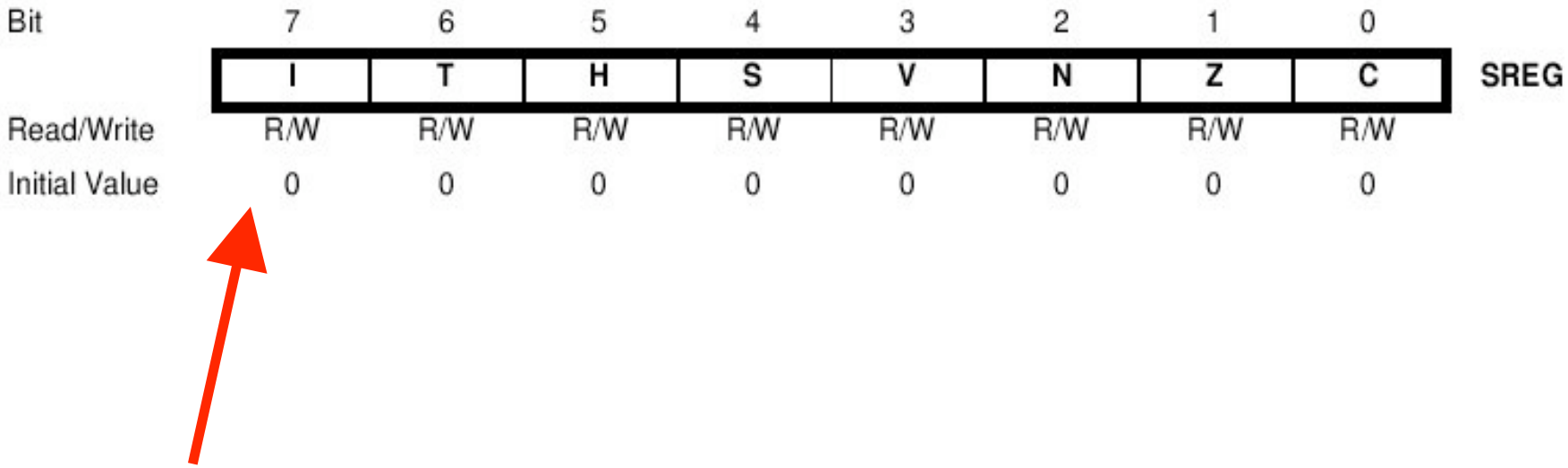
Atmel Mega8

Status register

- Many machine instructions affect the state of this register



Mega8 Status Register

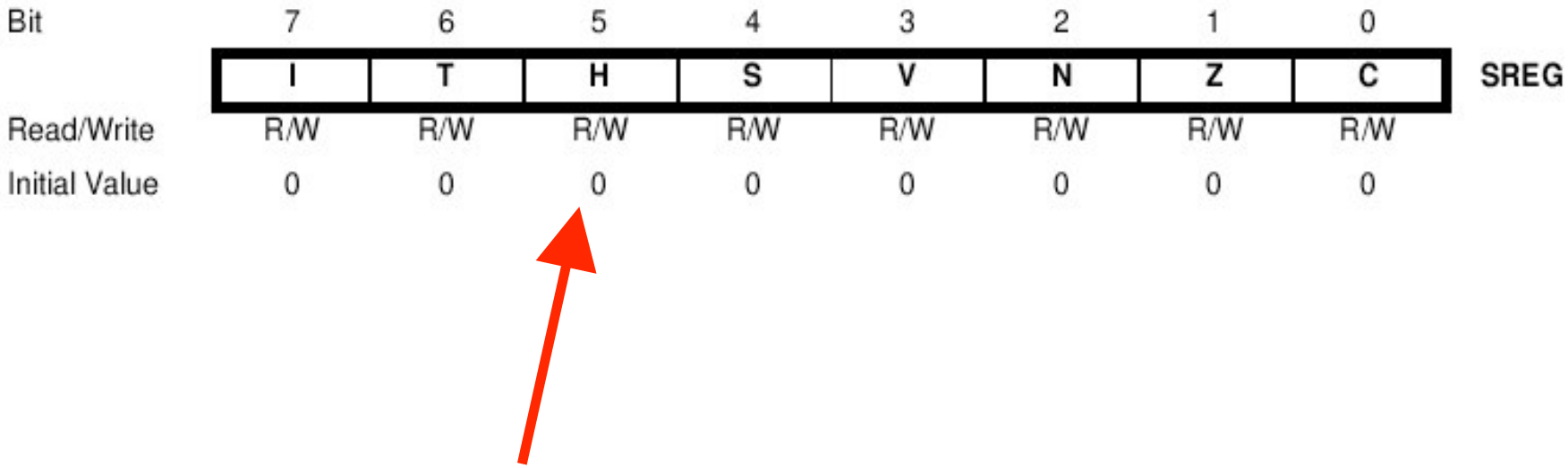


Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Interrupt enable

- If ‘1’, the currently executing program can be interrupted by another event (e.g., a byte arriving through the serial port)

Mega8 Status Register

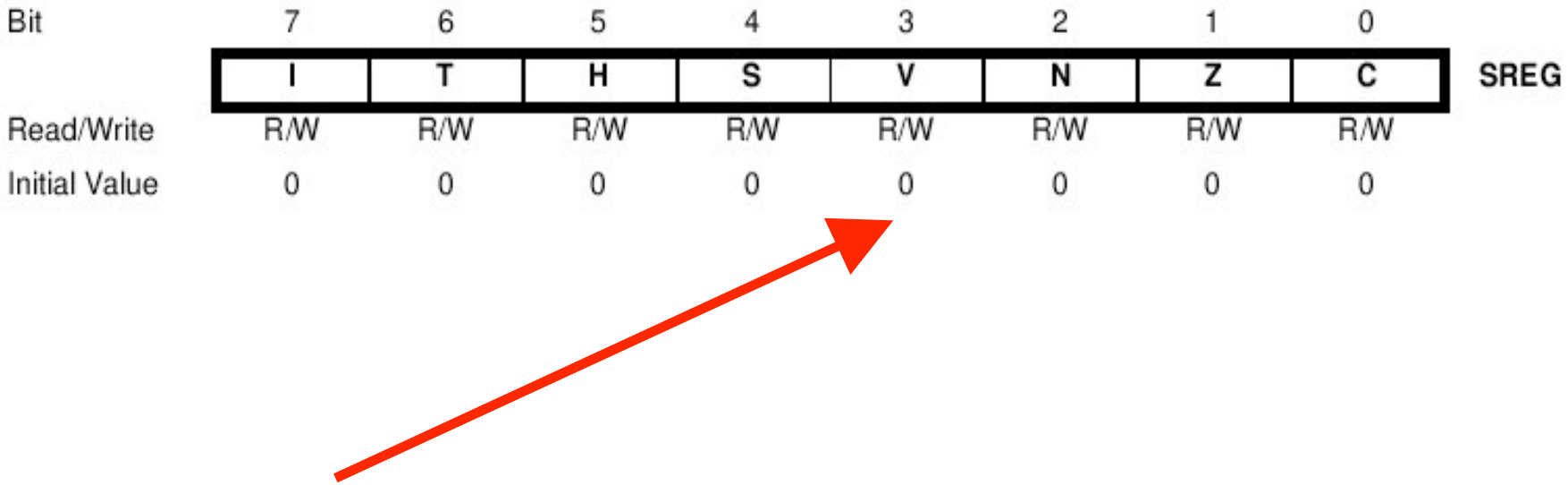


Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

Half carry flag

- Set if an arithmetic operation resulted in a carry from the first nybble to the next

Mega8 Status Register

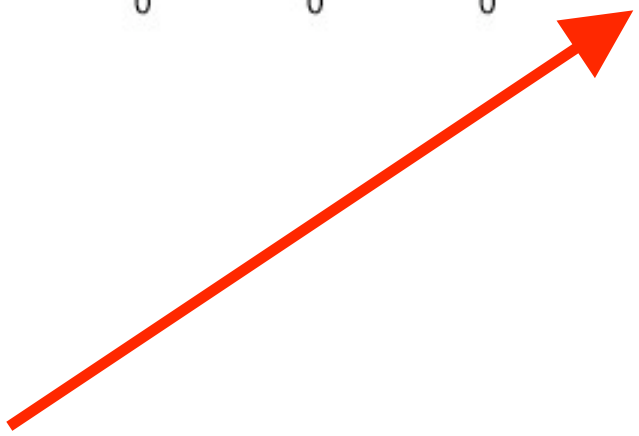


Two's complement overflow flag

- Set if an arithmetic operation resulted in an overflow in two's complement (e.g., incrementing an 8-bit number whose value is 127)

Mega8 Status Register

Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

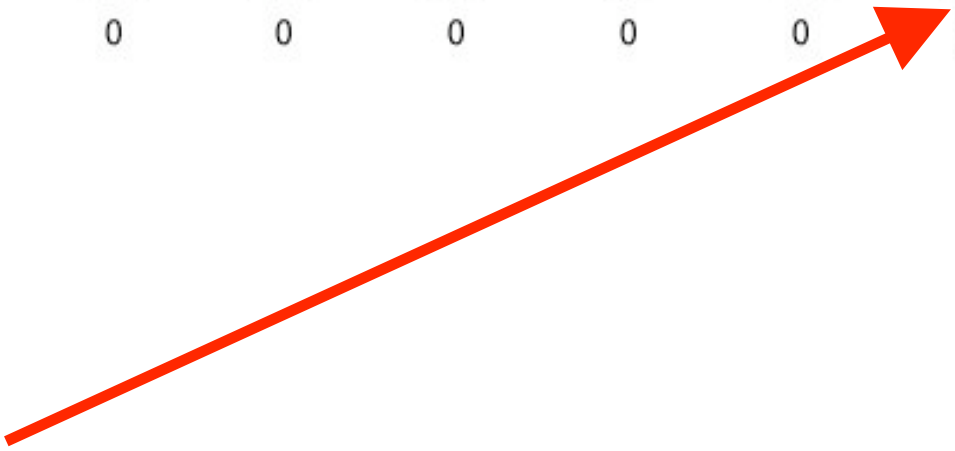


Negative flag

- Set if an arithmetic operation resulted in a negative value

Mega8 Status Register

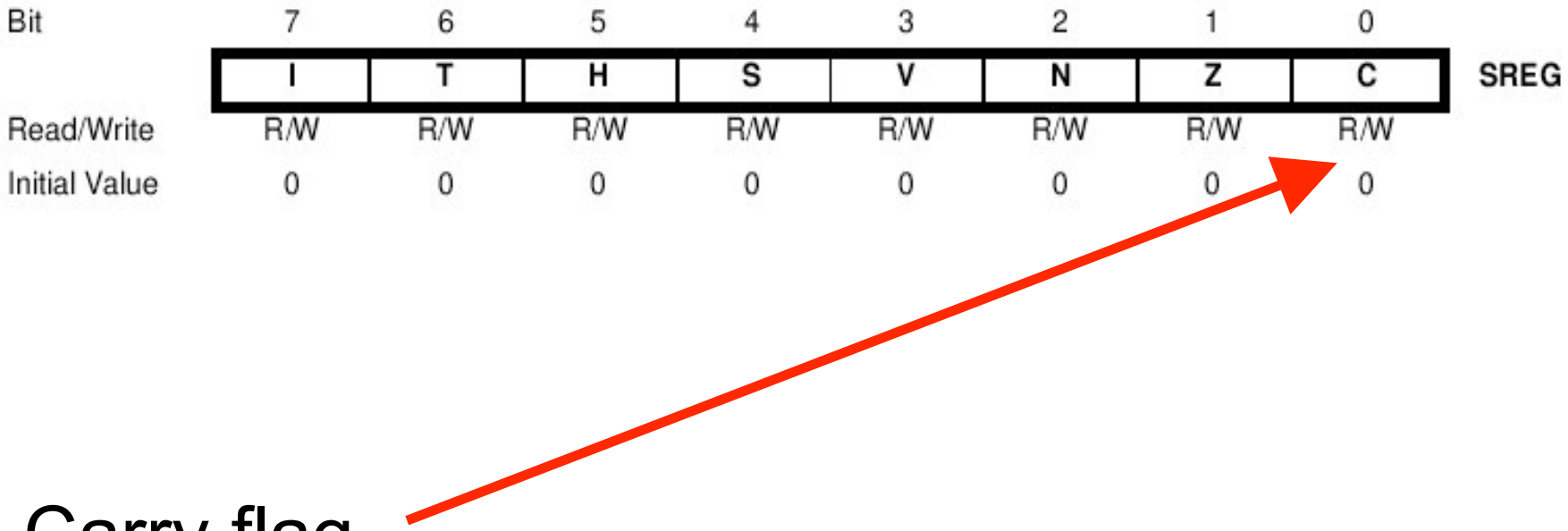
Bit	7	6	5	4	3	2	1	0	
	I	T	H	S	V	N	Z	C	SREG
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	



Zero flag

- Set if an arithmetic operation resulted in a value of zero

Mega8 Status Register



Carry flag

- Set if an arithmetic operation resulted in a carry (with an unsigned value)

Some Mega8 Memory Operations

LDS Rd, k

- Load SRAM memory location k into register Rd
- Rd <- (k)

STS Rd, k

- Store value of Rd into SRAM location k
- (k) <- Rd

Some Mega8 Memory Operations

LD Rd, Ry

- Load SRAM memory location indicated by Ry into register Rd
- Rd <- (Ry)

ST Rd, Ry

- Store value of Rd into SRAM location indicated by the value of Ry
- (Ry) <- Rd

Some Mega8 Arithmetic and Logical Instructions

ADD Rd, Rr

- Rd and Rr are registers
- Operation: Rd <- Rd + Rr
- Also affects status register (zero, carry, etc.)

ADC Rd, Rr

- Add with carry
- Rd <- Rd + Rr + C

Some Mega8 Arithmetic and Logical Instructions

NEG Rd: take the two's complement of Rd

AND Rd, Rr: bit-wise AND with a register

ANDI Rd, K: bit-wise AND with a constant

EOR Rd, Rr: bit-wise XOR

INC Rd: increment Rd

MUL Rd, Rr: multiply Rd and Rr (unsigned)

MULS Rd, Rd: multiply (signed)

Some Mega8 Test Instructions

CP Rd, Rr

- Compare Rd with Rr
- Alters the status register

TST Rd

- Test for zero or minus
- Alters the status register

Some Program Flow Instructions

RJMP k

- Change the program counter by k+1
- PC <- PC + k + 1

BRCS k

- Branch if carry set
- If C==1 then PC <- PC + k + 1

Connecting Assembly Language to C

- Our C compiler is responsible for translating our code into Assembly Language
- Today, we rarely program in Assembly Language
  - Embedded systems are a common exception
  - Also: it is useful in some cases to view the assembly code generated by the compiler

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

Last Time

- Review of midterm solutions
- Computer architecture
  - General-purpose vs special-purpose registers
  - Data bus
  - Instruction decoder
  - Function of a compiler

Today

- Complete computer architecture overview
- I/O systems
  - Polling
  - Interrupts
  - Direct memory access

Administrivia

Lab 3 due today

- Demonstration
- Group reports: hardcopy, postscript, or pdf
- Personal reports: raw text only!

Research Experiences for Undergraduate Site: Embedded Machine Learning Systems

- www-symbiotic.cs.ou.edu/reu/

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R1, R2
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Load the contents of memory location A into register 1

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Load the contents of memory location B into register 2

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Compare the contents of register 2 with those of register 1

This results in a change to the status register

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Branch if greater than or equal to will jump ahead 3 instructions if true

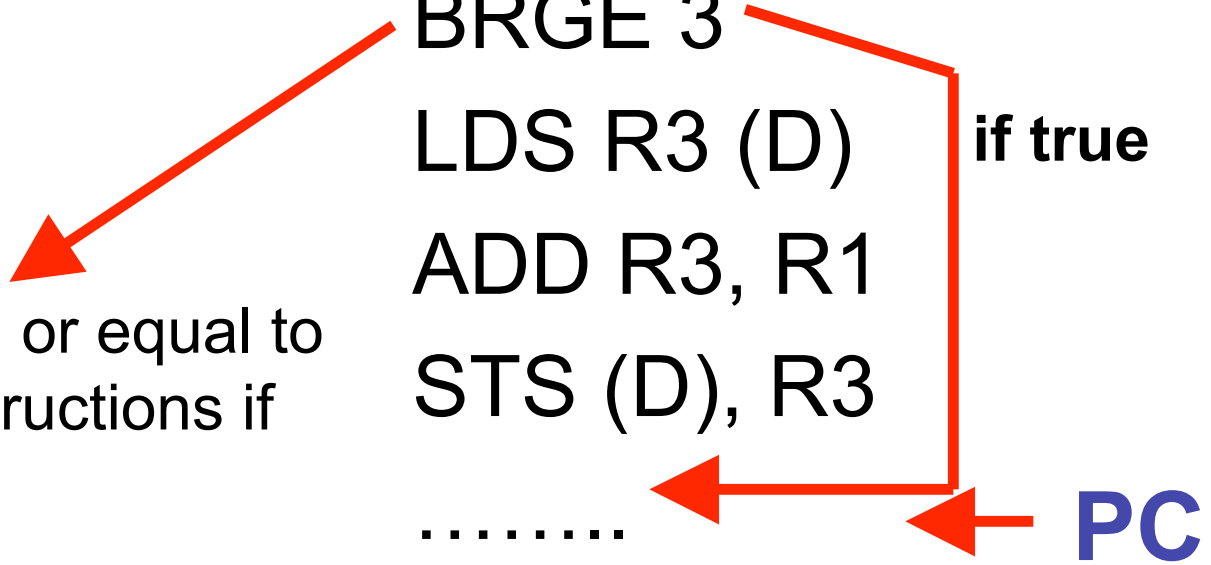
An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```



Branch if greater than or equal to will jump ahead 3 instructions if true

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

Not true: execute the next instruction

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

if not true

PC

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Load the contents of memory location D into register 3

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

Add the values in registers 1 and 3 and store the result in register 3

The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1   <- PC
STS (D), R3
……..
```

An Example

A C code snippet:

```
if(A > B) {
    D += A;
}
```

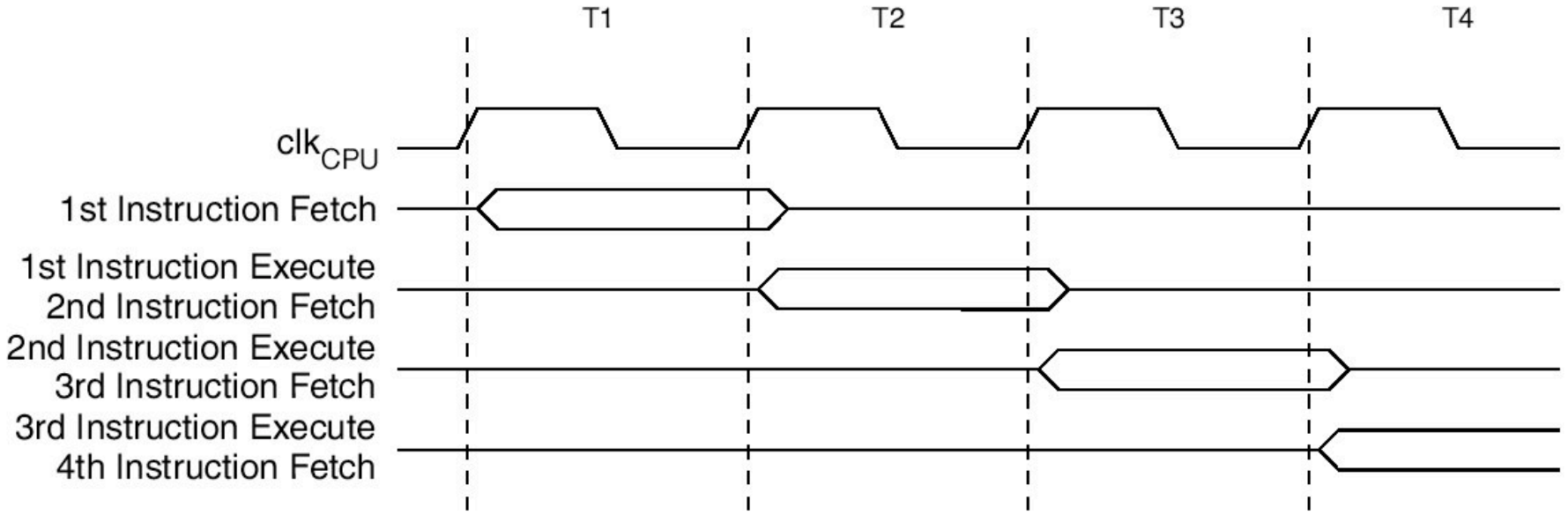
The Assembly :

```
LDS R1 (A)
LDS R2 (B)
CP R2, R1
BRGE 3
LDS R3 (D)
ADD R3, R1
STS (D), R3
……..
```

PC

Store the value in register 3 back to memory location D

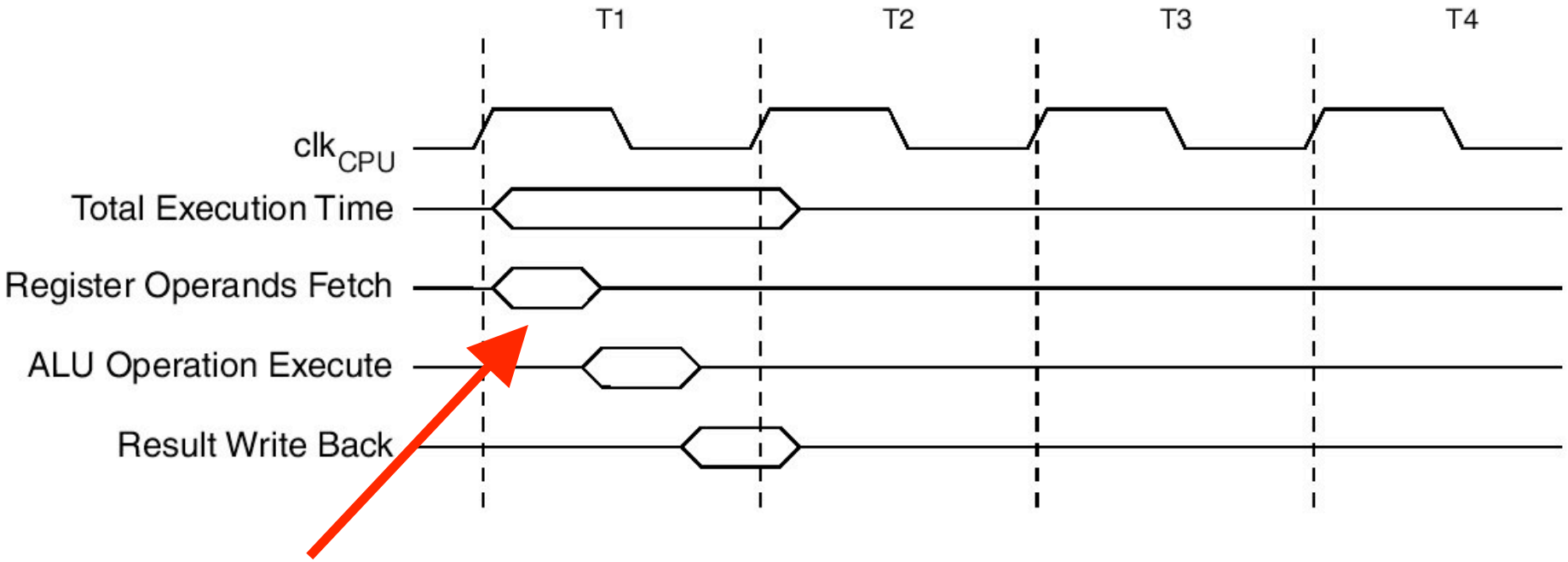
Instruction Fetch/Execution Cycle



From Atmel Mega8 spec

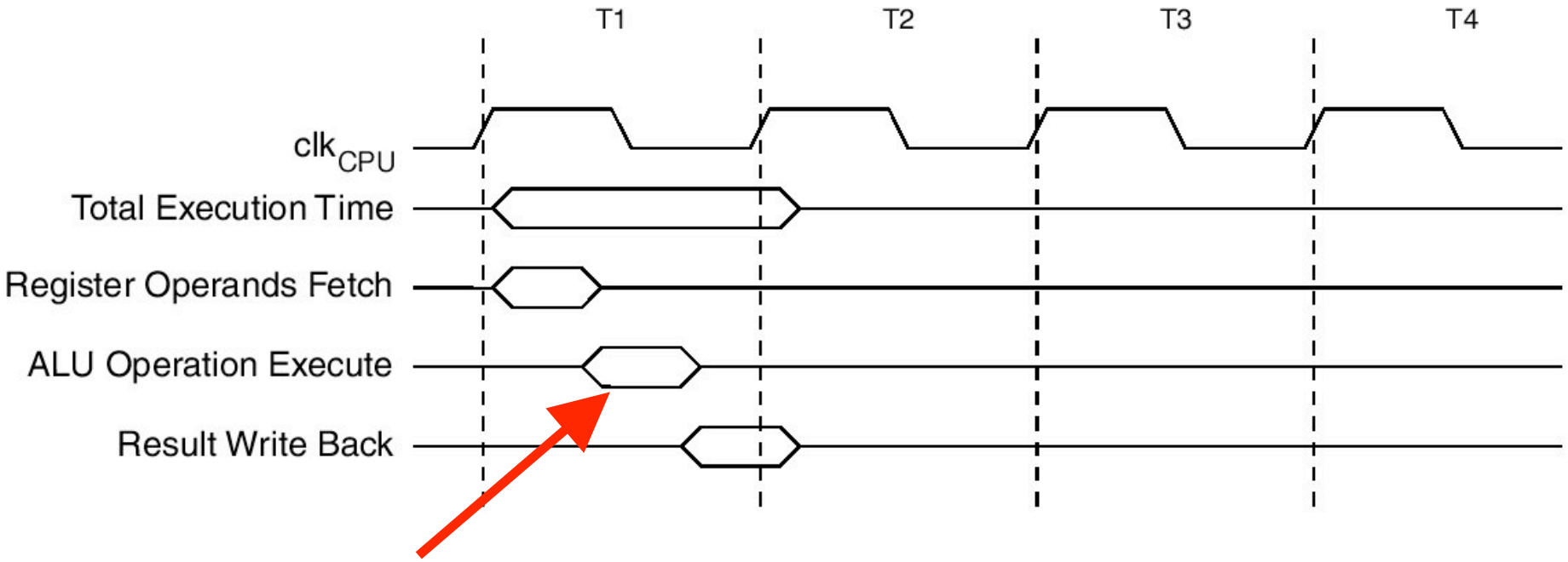
- While one instruction is being executed, the next is already being fetched from memory
- In many cases: each step happens on a single clock cycle

Instruction Execution Cycle



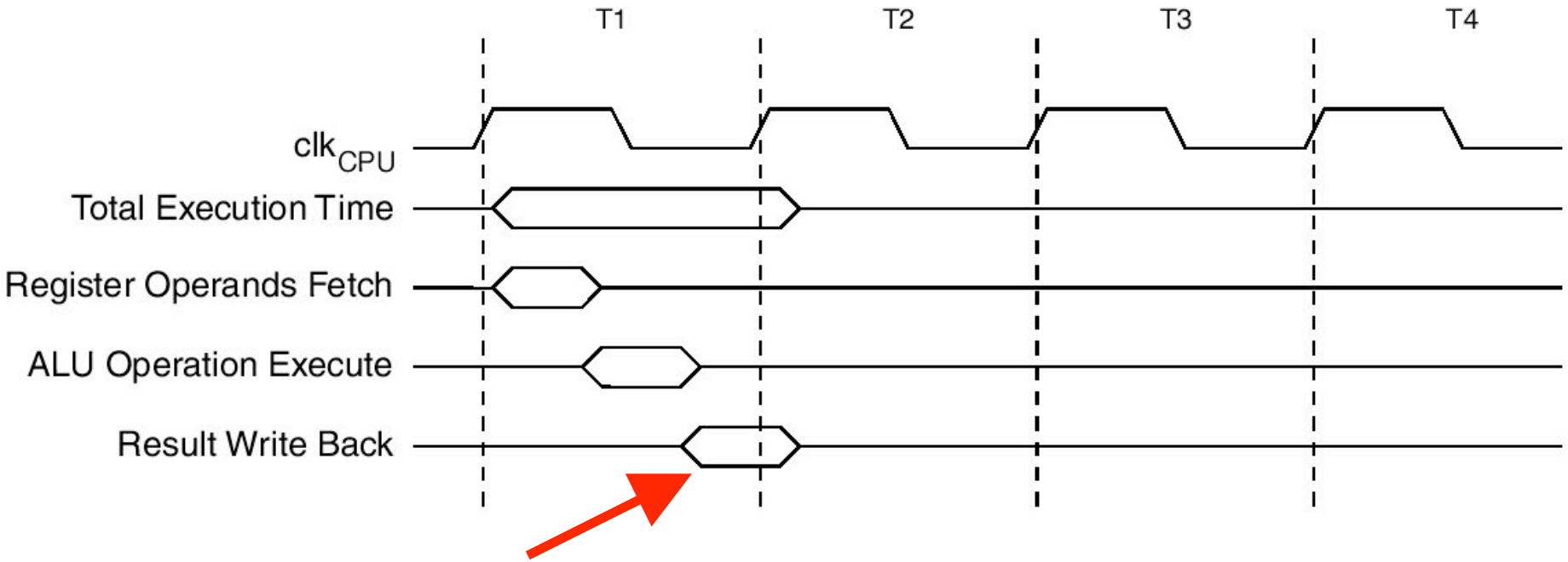
Address the registers and wait for the values to become available

Instruction Execution Cycle



Perform the operation dictated by the instruction

Instruction Execution Cycle



Result stored in destination register

Status register state changed

Instruction Representation

How are the individual instructions represented in the program memory?

Instruction Representation

How are the individual instructions represented in the program memory?

- As 16-bit binary numbers
- Each instruction (with parameters) is a unique binary number

Instruction Representation: An Example

Add with carry:

Assembly language: `ADC Rd, Rr`

Effect: Rd <- Rd + Rr + C

16-bit opcode:

0 0 0 1 1 1 r d d d d d r r r r

Instruction Representation: An Example

Assembly language: `ADC Rd, Rr`

Effect: Rd <- Rd + Rr + C

16-bit opcode:

0 0 0 1 1 1 r d d d d d r r r r

Specifies Rr

Instruction Representation: An Example

Assembly language: `ADC Rd, Rr`

Effect: Rd <- Rd + Rr + C

16-bit opcode:

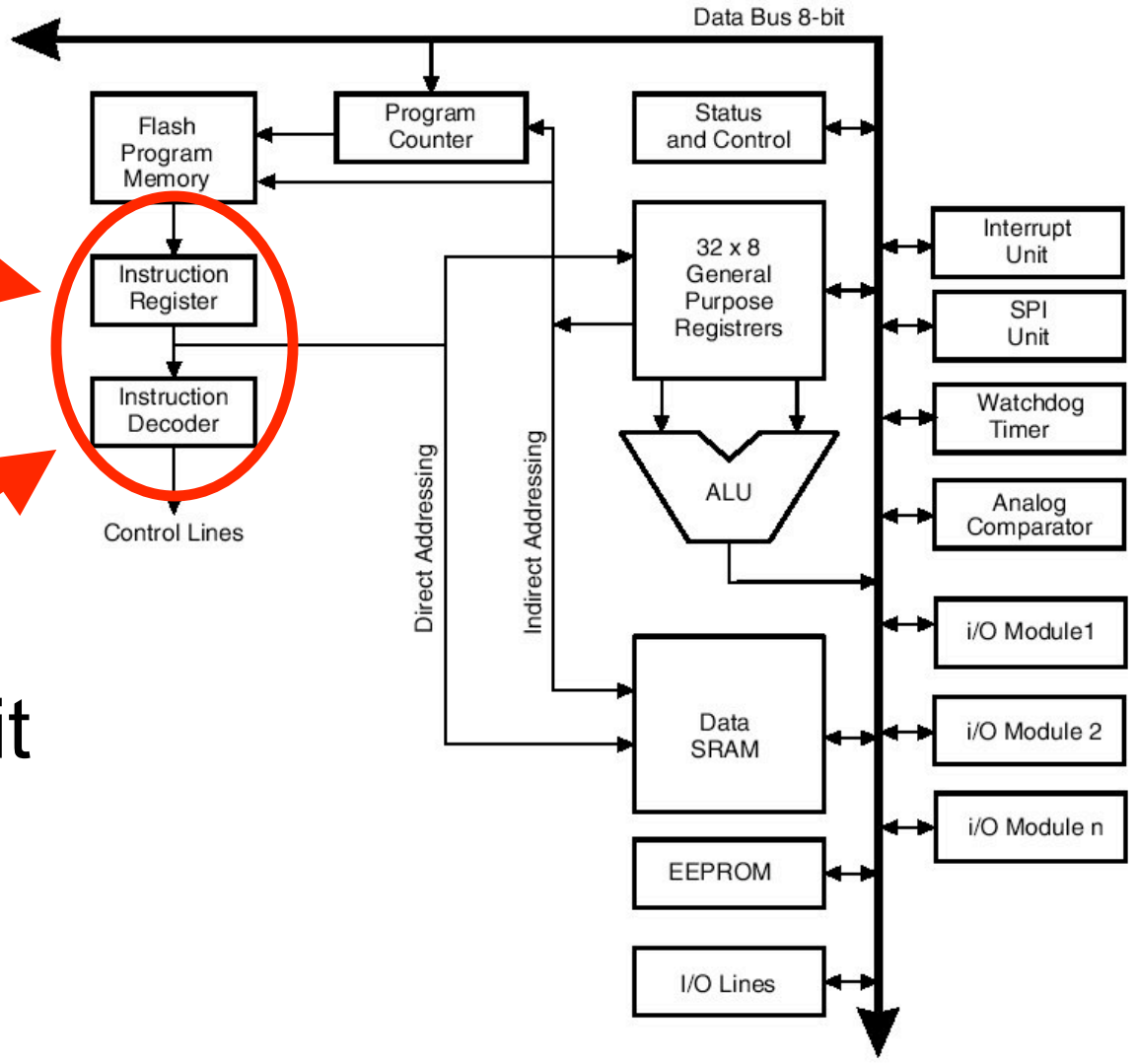
0 0 0 1 1 1 r d d d d d r r r r

Specifies Rd

Atmel Mega8: Decoding Instructions

Instruction register

Instruction decoder circuit



Instruction Decoding

- What does the circuit look like that would recognize this ADC instruction?
- How are the parameters distributed across the rest of the processor?

Instruction Representation: An Example II

Add without carry:

Assembly language: `ADD Rd, Rr`

Effect: Rd <- Rd + Rr

16-bit opcode:

0 0 0 0 1 1 r d d d d d r r r r

Instruction Representation: An Example III

Compare:

Assembly language: `CP Rd, Rr`

Effect: Rd - Rr

16-bit opcode:

0 0 0 1 0 1 r d d d d d r r r r

Instruction Representation: An Example IV

Arithmetic Shift Right:

Assembly language: `ASR Rd`

Effect: Rd <- “Rd >> 1”

16-bit opcode:

1 0 0 1 0 1 0 d d d d d 0 1 0 1

Next Time

- Input/Ouput Techniques
- Reading (for next week):
  - ESP Chapters 3 and 4