CSEP567: Design and Implementation of Digital Systems– Focus on Embedded Design

- Course staff:
  - Bruce Hemingway and Waylon Brunette, with Chris Grand
- Course web:
  - http://www.cs.washington.edu/education/courses/csep567/05au/
  - My office: CSE 464 Allen Center, 206 543-6274
- Today
  - Class administration, overview of course web, and logistics
  - What is an embedded system?
  - What will we be doing in this class?

Highlights:

- -- we'll be reading hand-outs and papers from various sources.
- -- The course work will be built around the Atmel AVR microprocessor .
- -- Tools are GCC and WinAVR.
  -- Languages are assembly and C.
- -- Laboratory assignments will focus on experiments with a microprocessor, and continue with the use of wireless motes.
- -- Lecture-discussion for an hour or so, then into the lab for the rest of the evening.
- -- No hardware experience required.

Labs:

- Lab 1- Atmel AVR Microprocessor circuit
- Lab 2 Write a short assembly program
- Lab 3- First C program– a counter
- Lab 4- LCD for debugging
- Lab 5- Color spaces
- Lab 6- Pulse Width Modulation
- Lab 7- SPI/ GUI interface
- Labs 8 and 9– TinyOS experiments

Introduction: The Digital Age

- Computing is in its infancy
  - Processing power
    - Doubles every 18 months
    - Factor of 100 / decade
  - Disk capacity
    - Doubles every 12 months
    - Factor of 1000 / decade
  - Optical fiber transmission capacity
    - Doubles every 9 months
    - Factor of 10,000 / decade
- The bases are mathematics and switches
  - How did we get here?

1854

All computers use Boolean algebra

- George Boole
  - Boolean algebra
- Number system with 2 values
  - 0/1 ⇔ false/true
  - Do math on logic statements
  - 3 operations (NOT, AND, OR)

NOT

A	Out
0	1
1	0

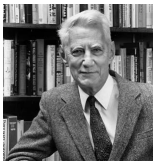
AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

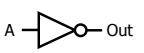
OR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

1938

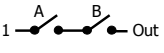


- Claude Shannon
  - Implemented Boolean algebra using switches
  - Described information using binary digits (bits)



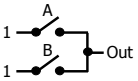
NOT

A	Out
0	1
1	0



AND

A	B	Out
0	0	0
0	1	0
1	0	0
1	1	1

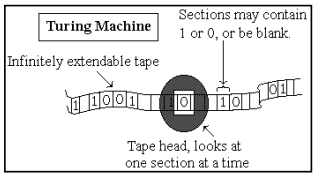


OR

A	B	Out
0	0	0
0	1	1
1	0	1
1	1	1

1937

- Alan Turing
  - Turing Machines
- Simple computer model
  - Can something be computed?



Also pioneered artificial intelligence

1945

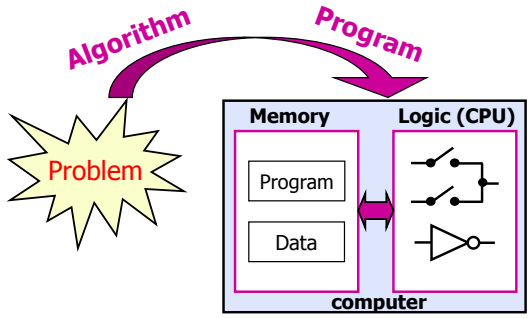
- John von Neumann
  - First stored computer program
- A sequence of operations
  - Read from memory
  - Operate using logic gates
  - Store result into memory

Other contributions:
Quantum Mechanics
Cellular Automata
Game Theory

Stored Programs = Software

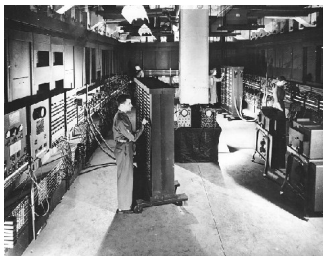
Bill Gates and Paul Allen, Lakeside, 1968

Hardware + Software



1946

- ENIAC…the first computer
  - Vacuum tubes for switches



1000x faster than anything before...

19,000 tubes

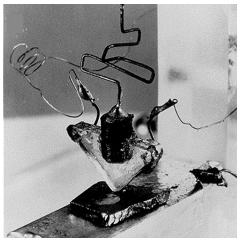
200 kilowatts

357 multiplies per second

1947

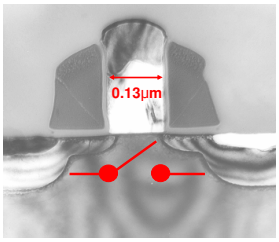
- Bardeen, Brattain, Shockley invent the transistor

1947



Nobel Prize, 1956

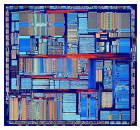
2000



Courtesy Mark Bohr, Intel

1958

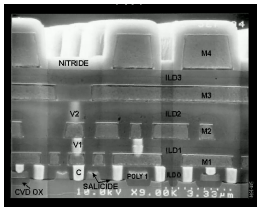
Pentium



- Kilby and Noyce invent the integrated circuit

2000

1958

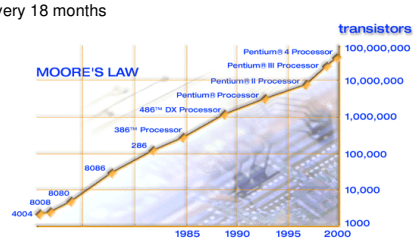


Nobel Prize, 2000

Courtesy Yan Borodovsky, Intel

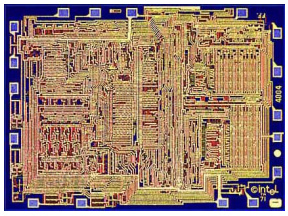
1965

- Gordon Moore
  - Moore's Law: The transistor density of silicon chips doubles every 18 months

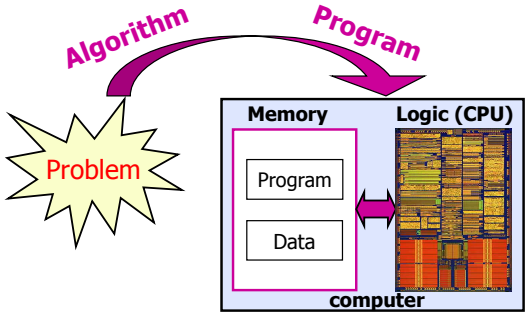


1971

- Ted Hoff invents the microprocessor
  - Intel 4004
    - 2,300 transistors
    - 3 mm by 4 mm
    - As powerful as the ENIAC



Hardware + Software + Technology



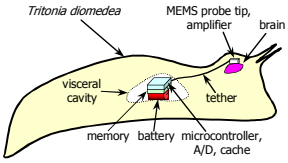
1977 and 1981

- Apple II and IBM PC
  - The first microcomputers

[13]

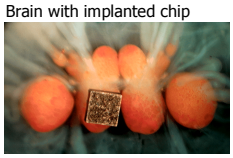
A modern example

- Goal: Interface a computer to an animal brain
  - Measure brain signals in intact animals



Tritonia and seapen

Brain with implanted chip



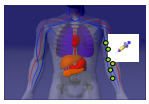
Courtesy Jim Beck and Russell Wyeth

More modern examples

- Computing everywhere
  - Wireless/wired networking
  - Wearable devices
  - Smart sensors







Embedded system – from the web

- Definitions
  - A device not independently programmable by the user.
  - Specialized computing devices that are not deployed as general purpose computers.
  - A specialized computer system which is dedicated to a specific task.
  - An embedded system is preprogrammed to perform a narrow range of functions with minimal end user or operator intervention.

Embedded system – from the web

- Examples
  - Virtually all appliances that have a digital interface -- watches, microwaves, VCRs, cars -- utilize embedded systems.
  - A computer system dedicated to controlling some non-computing hardware, like a washing machine, a car engine or a missile.
  - Examples of embedded systems are medical equipment and manufacturing equipment.
  - While most consumers aren't aware that they exist, they are extremely common, ranging from industrial systems to VCRs and many net devices.

What is an embedded system?

- Different than a desktop system
  - Fixed or semi-fixed functionality (not user programmable)
  - Different human interfaces than screen, keyboard, mouse, audio
  - Usually has sensors and actuators for interface to physical world
  - May have stringent real-time requirements
- It may:
  - Replace discrete logic circuits
  - Replace analog circuits
  - Provide feature implementation path
  - Make maintenance easier
  - Protect intellectual property
  - Improve mechanical performance

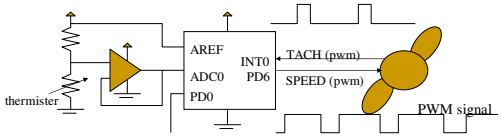
What do these differences imply?

- Less emphasis on
  - Graphical user interface
  - Dynamic linking and loading
  - Virtual memory, protection modes
  - Disks and file systems
  - Processes
- More emphasis on
  - Real-time support, interrupts (very small OS, if we're lucky)
  - Tasks (threads)
  - Task communication primitives
  - General-purpose input/output
  - Analog-digital/digital-analog converters
  - Timers
  - Event capture
  - Pulse-width modulation
  - Built-in communication protocols

What is an embedded system? (cont'd)

- Figures of merit for embedded systems
  - Reliability – it should never crash
  - Safety – controls things that move and can harm/kill a person
  - Power consumption – may run on limited power supply
  - Cost – engineering cost, manufacturing cost, schedule tradeoffs
  - Product life cycle – maintainability, upgradeability, serviceability
  - Performance – real-time requirements, power budget

Example: a temperature controller



Task: Tachometer (external interrupt)
```
now = getTime();
period = then - now; //overflow?
then = now;
return;
```

Task: TempControl (periodic, soft constraint)
```
if (Temp > setpoint) Thi++;
if (Temp < setpoint) Thi--;
if (period<min || period>max) PD0 = 1;
```

Task: FanPWM (periodic, hard constraint)
```
count++;
if (count == 0) PD6 = 1;
if (count > Thi) PD6 = 0;
return;
```

Task: Main
```
Thi = 0;
setup timer for 1ms interrupt;
setup timer for 100ms interrupt;
while (1) ;
```


Capacity

- Assume:
  - 8 MHz processor @ one instruction/cycle
  - Assume fan runs between 30Hz and 60Hz
  - Assume 256ms period on speed control PWM, with 1ms resolution.
- What percent of the the available cycles are used for the temperature controller?
  - [total instructions in one second] / (8MInstr/sec)
- How much RAM do you need?
- How much ROM?

Resource analysis of temp controller

Task: Tachometer (external interrupt)
```
now = getTime();
period = then - now; //overflow?
then = now;
return;
```

Task: TempControl (periodic, soft constraint)
```
if (Temp > setpoint) Thi++;
if (Temp < setpoint) Thi--;
if (period<min || period>max) GP4 = 1;
```

Task: FanPWM (periodic, hard constraint)
```
count++;
if (count == 0) GP0 = 1;
if (count > Thi) GP0 = 0;
return;
```

Task: Main
```
Thi = 0;
setup timer for 1ms interrupt;
setup timer for 100ms interrupt;
while (1) ;
```

Task	ROM	RAM	Instructions/Sec
Tach	~4	2 (period, then)	4 * 60 = 240
FanPWM	~8	1 (count)	8 * 1000 = 8000
TempControl	~10	1 (THI)	10 * 2 = 20

Total Instructions/Sec = 8260, at 8MIPS, that's only 0.1% utilization!
Other resources? local variables, stack

Microprocessors

- Arbitrary computations
  - Arbitrary control structures
  - Arbitrary data structures
  - Specify function at high-level and use compilers and debuggers
- Microprocessors can lower hardware costs
  - If function requires too much logic when implemented with gates/FFs
    - Operations are too complex, better broken down as instructions
    - Lots of data manipulation (memory)
  - If function does not require higher performance of customized logic
    - Ever-increasing performance of processors puts more and more applications in this category
    - Minimize the amount of external logic

Microprocessor basics

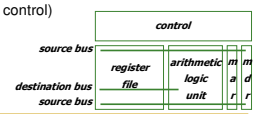
- Composed of three parts
  - Data-path: data manipulation and storage
  - Control: determines sequence of actions executed in data-path and interactions to be had with environment
  - Interface: signals seen by the environment of the processor
- Instruction execution engine: fetch/execute cycle
  - Flow of control determined by modifications to program counter
  - Instruction classes:
    - Data: move, arithmetic and logical operations
    - Control: branch, loop, subroutine call
    - Interface: load, store from external memory

Microprocessor basics (cont'd)

- Can implement arbitrary state machine with auxiliary data-path
  - Control instructions implement state diagram
  - Registers and ALUs act as data storage and manipulation
  - Interaction with the environment through memory interface
  - How are individual signal wires sensed and controlled?

Microprocessor organization

- Controller
  - Inputs: from ALU (conditions), instruction read from memory
  - Outputs: select inputs for registers, ALU operations, read/write to memory
- Data-path
  - Register file to hold data
  - Arithmetic logic unit to manipulate data
  - Program counter (to implement relative jumps and increments)
- Interface
  - Data to/from memory (address and data registers in data path)
  - Read/write signals to memory (from control)



General-purpose processor

- Programmed by user
- New applications are developed routinely
- General-purpose
  - Must handle a wide ranging variety of applications
- Interacts with environment through memory
  - All devices communicate through memory data
  - DMA operations between disk and I/O devices
  - Dual-ported memory (e.g., display screen)
  - Generally, oblivious to passage of time

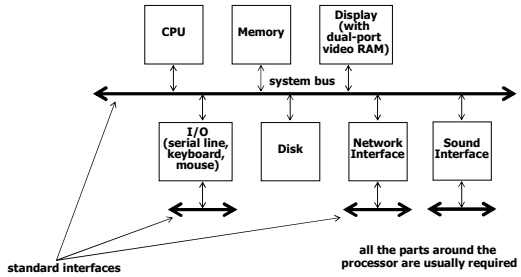
Embedded processor

- Typically programmed once by manufacturer of system
  - Rarely does the user load new software
- Executes a single program (or a limited suite) with few parameters
- Task-specific
  - Can be optimized for a specific application
- Interacts with environment in many ways
  - Direct sensing and control of signal wires
  - Communication protocols to environment and other devices
  - Real-time interactions and constraints
  - Power-saving modes of operation to conserve battery power

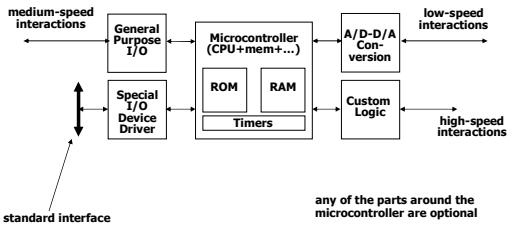
Why embedded processors?

- High overhead in building a general-purpose system
  - Storing/loading programs
  - Operating system manages running of programs and access to data
  - Shared system resources (e.g., system bus, large memory)
  - Many parts
    - Communication through shared memory/bus
    - Each I/O device often requires its own separate hardware unit
- Optimization opportunities
  - As much hardware as necessary for application
    - Cheaper, portable, lower-power systems
  - As much software as necessary for application
    - Doesn't require a complete OS, get a lot done with a smaller processor
  - Can integrate processor, memory, and I/O devices on to a single chip

Typical general-purpose architecture



Typical task-specific architecture



How does this change things?

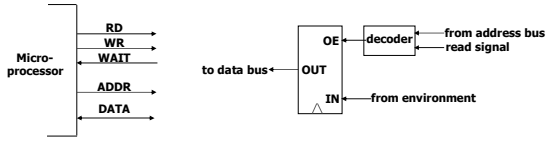
- Sense and control of environment
  - Processor must be able to “read” and “write” individual wires
  - Controls I/O interfaces directly
- Measurement of time
  - Many applications require precise spacing of events in time
  - Reaction times to external stimuli may be constrained
- Communication
  - Protocols must be implemented by processor
  - Integrate I/O device or emulate in software
  - Capability of using external device when necessary

Interactions with the environment

- Basic processor only has address and data busses to memory
- Inputs are read from memory
- Outputs are written to memory
- Thus, for a processor to sense/control signal wires in the environment they must be made to appear as memory bits
  - How do we make wires look like memory?

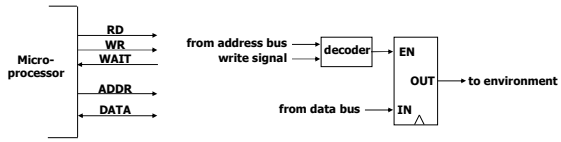
Sensing external signals

- Map external wire to a bit in the address space of the processor
- External register or latch buffers values coming from environment
  - Map register into address space
    - Decoder selects register for reading
  - Output enable (OE) to get value on to data bus
    - Lets many registers use the same data bus



Controlling external signals

- Map external wire to a bit in the address space of the processor
- Connect output of memory-mapped register to environment
  - Map register into address space
    - Decoder selects register for writing (holds value indefinitely)
  - Input enable (EN) to take value from data bus
    - Lets many registers use the same data bus



Time and instruction execution

- Keep track of detailed timing of each instruction's execution
  - Highly dependent on code
  - Hard to use compilers
  - Not enough control over code generation
  - Interactions with caches/instruction-buffers
- Loops to implement delays
  - Keep track of time in counters
  - Keeps processor busy counting and not doing other useful things
- Timer
  - Take differences between measurements at different points in code
  - Keeps running even if processor is idle to save power
  - An independent “co-processor” to main processor

Time measurement via parallel timers

- Separate and parallel counting unit(s)
  - Co-processor to microprocessor
  - Does not require microprocessor intervention
  - May be a simple counter or a more featured real-time clock
  - Alarms can be set to generate interrupts
- More interesting timer units
  - Self reloading timers for regular interrupts
  - Pre-scaling for measuring larger times
  - Started by external events

Input/output events

- Input capture
  - Record time when input event occured
  - Can be used in later handling of event
- Output compare
  - Set output event to happen at a point in the future
  - Reactive outputs
    - e.g., set output to happen a pre-defined time after some input
  - Processor can go on to do other things in the meantime

System bus based communication

- Extend address/data bus outside of chip
- Use specialized devices to implement communication protocol
- Map devices and their registers to memory locations
- Read/write data to receive/send buffers in shared memory or device
- Poll registers for status of communication
- Wait for interrupt from device on interesting events
  - Send completed
  - Receive occurred

Support for communication protocols

- Built-in device drivers
  - For common communication protocols
    - e.g., RS232, IrDA, USB, Bluetooth, etc.
  - Serial-line protocols most common as they require fewer pins
- Serial-line controller
  - Special registers in memory space for interaction
  - May use timer unit(s) to generate timing events
    - For spacing of bits on signal wire
    - For sampling rate
- Increase level of integration
  - No external devices
  - May further eliminate need for shared memory or system bus

Microcontrollers

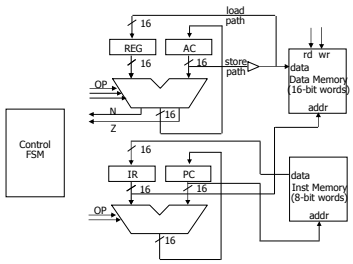
- Embedded processor with much more integrated on same chip
  - Processor core + co-processors + memory
  - ROM for program memory, RAM for data memory, special registers to interface to outside world
  - Parallel I/O ports to sense and control wires
  - Timer units to measure time in various ways
  - Communication subsystems to permit direct links to other devices

Microcontrollers (cont'd)

- Other features not usually found in general-purpose CPUs
  - Expanded interrupt handling capabilities
    - Multiple interrupts with priority and selective enable/disable
    - Automatic saving of context before handling interrupt
    - Interrupt vectoring to quickly jump to handlers
  - More instructions for bit manipulations
    - Support operations on bits (signal wires) rather than just words
- Integrated memory and support functions for cheaper system cost
  - Built-in EEPROM, Flash, and/or RAM
  - DRAM controller to handle refresh
  - Page-mode support for faster block transfers

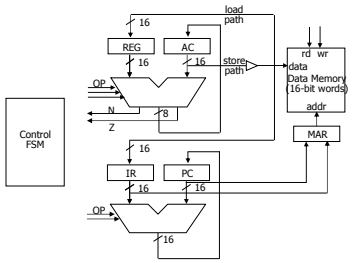
Block diagram of processor (Harvard)

- Register transfer view of Harvard architecture
  - Separate busses for instruction memory and data memory

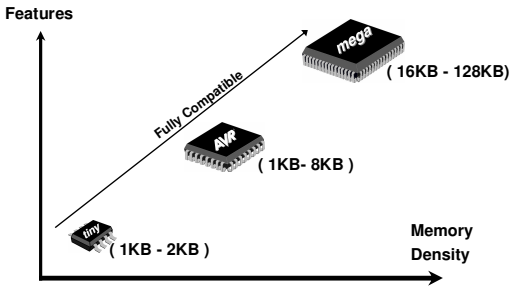


Block diagram of processor (Princeton)

- Register transfer view of Princeton architecture
  - Single unified bus for instructions, data, and I/O



The AVR Microcontroller Family



The AVR Microcontroller Family

	tiny11	tiny12	tiny15	tiny28
Pins	8	8	8	28/32
Flash	1 KB	1 KB	1 KB	2 KB
EEPROM	-	64 B	64 B	-
PWMs	-	-	1	1
ADC	-	-	4@10-bit	-

	S1200	S2323	S2343	S2313
Pins	20	8	8	20
Flash	1 KB	2 KB	2 KB	2 KB
SRAM	-	128 B	128 B	128 B
EEPROM	64 B	128 B	128 B	128 B
UART	-	-	-	1
PWM	-	-	-	1

	S4433	S8515	VC8534	S8535
Pins	28/32	40/44	48	40/44
Flash	4 KB	8 KB	8 KB	8 KB
SRAM	128 B	512 B	256 B	512 B
EEPROM	256 B	512 B	512 B	512 B
UART	1	1	-	1
PWM	1	2	-	2
ADC	6@10-bit	-	6@10-bit	8@10-bit
RTC	-	-	-	Yes

The AVR Microcontroller Family

	mega161	mega163	mega32	mega103
Pins	40/44	40/44	40/44	64
Flash	16 KB	16 KB	32 KB	128 KB
SRAM	1 KB	1 KB	2 KB	4 KB
EEPROM	512 B	512 B	1 KB	2 KB
U(S)ART	2	1	1	1
TWI	1	1	1	-
PWM	4	4	4	4
ADC	-	8@10-bit	8@10-bit	8@10-bit
RTC	Yes	Yes	Yes	Yes
JTAG/OCD	-	-	Yes	-
Self Program	Yes	Yes	Yes	-
HW MULT	Yes	Yes	Yes	-
Brown Out	Yes	Yes	Yes	-

	mega8	mega16	mega32	mega64	mega128
Pins	28/32	40/44	40/44	64	64
Flash	8 KB	16 KB	32 KB	64 KB	128 KB
SRAM	1 KB	1 KB	2 KB	4 KB	4 KB
EEPROM	512 B	512 B	1 KB	2 KB	4 KB
U(S)ART	1	1	1	2	2
TWI	1	1	1	1	1
PWM	3	4	4	8	8
ADC	8@10-bit	8@10-bit	8@10-bit	8@10-bit	8@10-bit
RTC	Yes	Yes	Yes	Yes	Yes
JTAG/OCD	-	Yes	Yes	Yes	Yes
Self Program	Yes	Yes	Yes	Yes	Yes
HW MULT	Yes	Yes	Yes	Yes	Yes
Brown Out	Yes	Yes	Yes	Yes	Yes

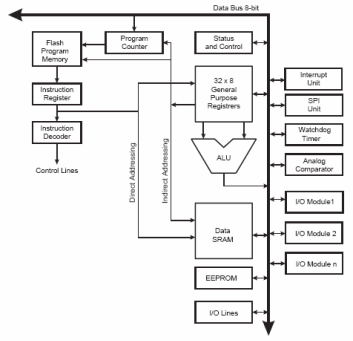
Microcontroller we will be using

- Atmel AVR Microcontroller (ATmega16) – 16 MIPS at 16 MHz
  - 8-bit microcontroller (8-bit data, 16-bit instructions) – RISC Architecture
    - 131 instructions (mostly single-cycle – on-chip 2-cycle multiplier)
    - 32 general-purpose registers
    - Internal and external interrupts
  - Memory
    - instruction (16KB Flash memory – read-while-write)
    - boot ROM (512 Byte EEPROM)
    - data (1K static RAM)
  - Timers/counters
    - 2 8-bit and 1 16-bit timer/counters with compare modes and prescalers
    - Real-time clock (32.768 kHz) with separate oscillator
    - Programmable watchdog timer
  - Serial communication interfaces
    - JTAG boundary-scan interface for programming/debugging
    - Programmable USART (universal synchronous/asynchronous receiver transmitter)
    - Two-wire serial interface (can emulate different communication protocols)
    - SPI serial port (serial peripheral interface)
  - Peripheral features
    - Four channels with support for pulse-width modulation
    - Analog-digital converter (8-channel, 10-bit)
    - Up to 32 general-purpose I/O pins (with interrupt support)
  - Six power saving modes

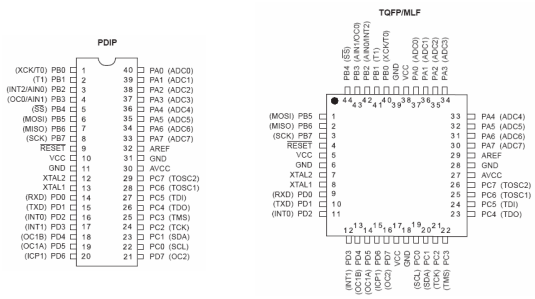
Why did we pick the ATmega16

- Modern microcontroller
- Easy to use C compiler
- Better performance/power than competitors
  - Microchip PIC
  - Motorola 68HC11
  - Intel 80C51
- Excellent support for 16-bit arithmetic operations
- A lot of registers that eliminate moves to and from SRAM
- Single cycle execution of most instructions
- Used in the UC Berkeley sensor mote (last two labs)

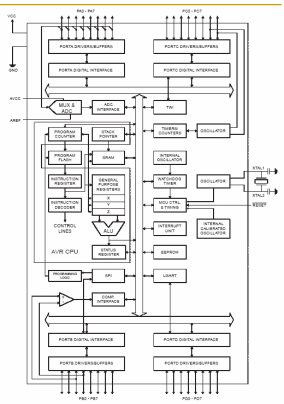
ATMega16 Overview



ATMega16 Pinouts

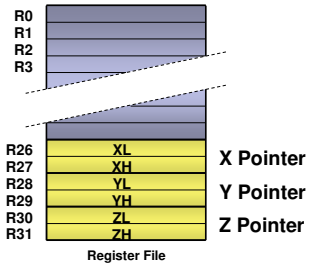


ATmega16 Internals



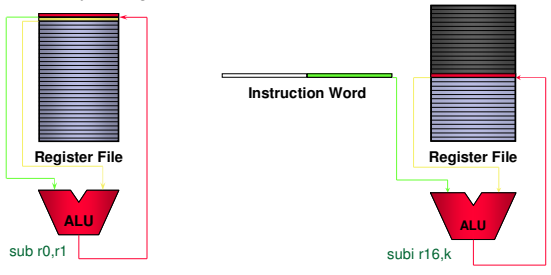
General-Purpose Working Registers

- 32 registers
- 6 are special
  - Used for addressing modes
- Addressed in regular memory space
  - Easier to use instructions

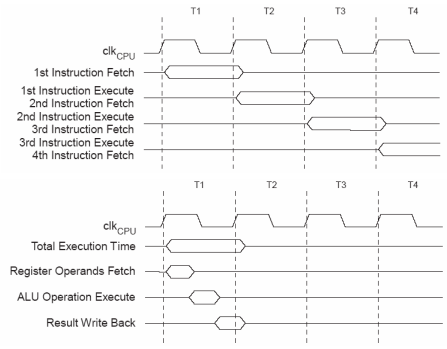


Operations on Register/Immediate Values

- One cycle register/immediate instructions



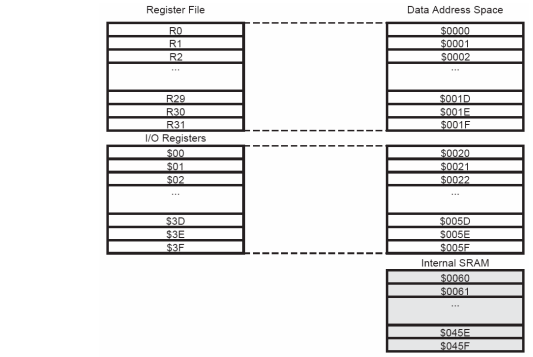
Instruction Timing



The Five Memory Areas

- General Purpose Register File = 32 B
- Flash Program Memory = 8 KB (≤ 8 MB)
- SRAM Data Memory = 1 KB (≤ 16 MB)
- I/O Memory = 64 B (≤ 64 B)
- EEPROM Data Memory = 512 B (≤ 16 MB)

Memory Map



Data SRAM −> Register File (RF)

- "LD Rd,<PTR>" Load indirect
- "LD Rd,<PTR>+" Load indirect with post-increment
- "LD Rd,-<PTR>" Load indirect with pre-decrement
- "LDD Rd,<PTR>+q" Load indirect with displacement (0-63)*

** PTR =X, Y or Z*

Data SRAM <− Register File (RF)

- "ST <PTR>,Rd" Store indirect
- "ST <PTR>+,Rd" Store indirect with post-increment
- "ST -<PTR>,Rd" Store indirect with pre-decrement
- "STD <PTR>+q,Rd" Store indirect with displacement (0-63) *

** PTR = X, Y or Z*

Data Transfer Program Memory −> RF

- "LDI" Load a register with an immediate value (1 Cycle) *
- "LPM" Transfer a byte from program Memory@Z to R0 (3 Cycles)
- "LPM Rd,Z" Transfer a byte from program Memory@Z to Rd (3 Cycles)
- "LPM Rd,Z+" As above but with post-increment of the Z pointer

** Works on R16 - R31*

Register File −> Register File

- "OUT" Transfer a byte from RF to I/O
- "IN" Transfer a byte from I/O to RF

- "MOV" Copy a register to another register
- "MOVW" Copy a register pair to another register pair. Aligned.

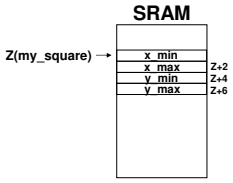
C-like Addressing Modes (1)

- Auto Increment/Decrement:
  - C Source:
    - unsigned char *var1, *var2;
    - *var1++ = *--var2;
  - Generated code:
    - LD R16,-X
    - ST Z+,R16

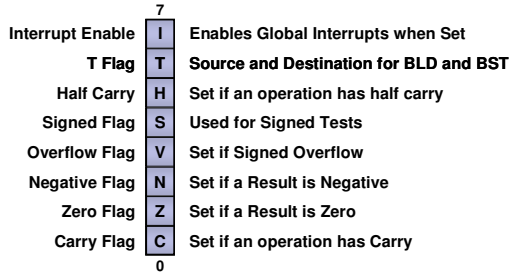
C-Like Addressing Modes (2)

- Indirect with displacement
- Efficient for accessing arrays and structs

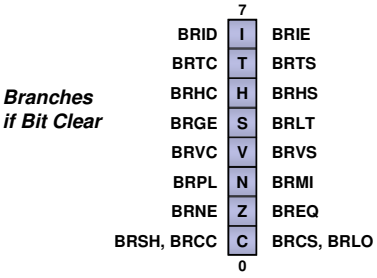
```
Struct square
{
    int x_min;
    int x_max;
    int y_min;
    int y_max;
}my_square;
```



The Status Register - SREG



Branch on SREG Settings



Branches if Bit Clear

Branches if Bit Set

A Small C Function

```
/* Return the maximum value of a table of 16 integers */

int max(int *array)
{
  char a;
  int maximum=-32768;

  for (a=0;a<16;a++)
     if (array[a]>maximum)
        maximum=array[a];
  return (maximum);
}
```


AVR Assembly output

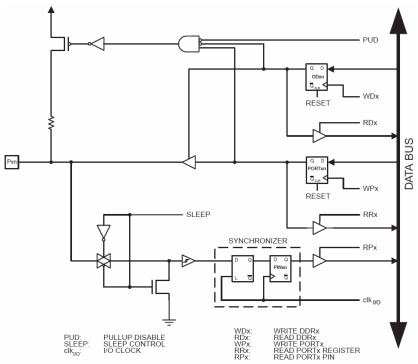
```
; 7.    for (a=0;a<16;a++)
        LDI     R18,LOW(0)
        LDI     R19,128
        CLR     R22
?0001:
        CPI     R22,LOW(16)
        BRCC    ?0000
; 8.      {
; 9.        if (array[a]>maximum)
        MOV     R30,R22
        CLR     R31
        LSL     R30
        ROL     R31
        ADD     R30,R16
        ADC     R31,R17
        LDD     R20,Z+0
        LDD     R21,Z+1
        CP      R18,R20
        CPC     R19,R21
        BRGE    ?0005
; 10.               maximum=array[a];
        MOV     R18,R20
        MOV     R19,R21
?0005:
        INC     R22
        RJMP    ?0001
?0000:
; 11.     }
; 12.     return (maximum);
        MOV     R16,R18
        MOV     R17,R19
; 13.   }
        RET
```

Code Size: 46 Bytes, Execution time: 335 cycles

I/O Ports General Features

- Push-pull drivers
- High current drive (sinks up to 40 mA)
- Pin-wise controlled pull-up resistors
- Pin-wise controlled data direction
- Fully synchronized inputs
- Three control/status bits per bit/pin

I/O Ports

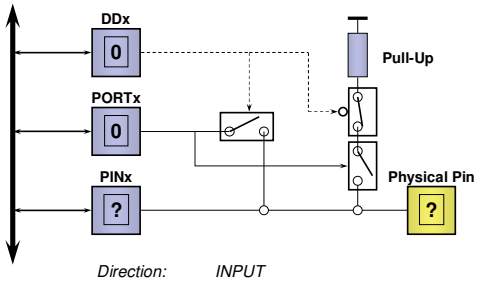


I/O Port Configurations

- 3 Control/Status Bits per Pin
  - DDx Data Direction Control Bit
  - PORTx Output Data or Pull-Up Control Bit
  - PINx Pin Level Bit

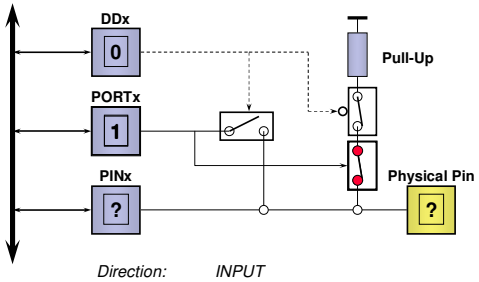
DDxn	PORTxn	PUD (in SFIOR)	I/O	Pull-up	Comment
0	0	X	Input	No	Tri-state (Hi-Z)
0	1	0	Input	Yes	Pxn will source current if ext. pulled low.
0	1	1	Input	No	Tri-state (Hi-Z)
1	0	X	Output	No	Output Low (Sink)
1	1	X	Output	No	Output High (Source)

Port is Input (Default Configuration)



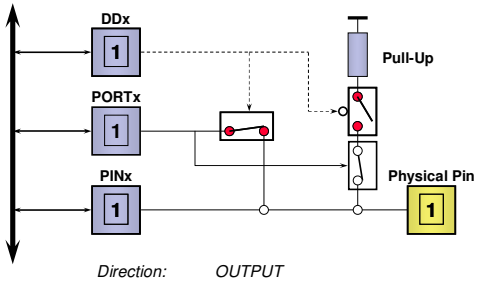
Direction: *INPUT*
Pull-Up: *OFF*

Port is Open-Collector Input (Switch On Pull-Up)



Direction: *INPUT*
Pull-Up: *ON*

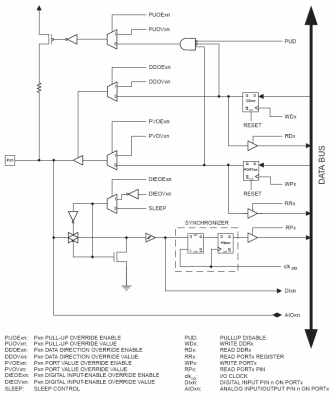
Port is Output



Direction: *OUTPUT*
Pull-Up: *OFF*

Alternate Port Functions

- Generalizing I/O ports so that they can be used by other I/O devices



PUOEn: Pxn PULL-UP OVERRIDE ENABLE
PUOVn: Pxn PULL-UP OVERRIDE VALUE
DDOEn: Pxn DATA DIRECTION OVERRIDE ENABLE
DDOVn: Pxn DATA DIRECTION OVERRIDE VALUE
PVOEn: Pxn PORT VALUE OVERRIDE ENABLE
PVOVn: Pxn PORT VALUE OVERRIDE VALUE
DIEOEn: Pxn DIGITAL INPUT-ENABLE OVERRIDE ENABLE
DIEOVn: Pxn DIGITAL INPUT-ENABLE OVERRIDE VALUE
SLEEP: SLEEP CONTROL

PUD: PULLUP DISABLE
WDx: WRITE DDRx
RDx: READ DDRx
RRx: READ PORTx REGISTER
WPx: WRITE PORTx
RPx: READ PORTx PIN
clkI/O: I/O CLOCK
DIxn: DIGITAL INPUT PIN n ON PORTx
AIOxn: ANALOG INPUT/OUTPUT PIN n ON PORTx

I/O Port Registers

Port A Data Register – PORTA

Bit	7	6	5	4	3	2	1	0	
	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	PORTA
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	

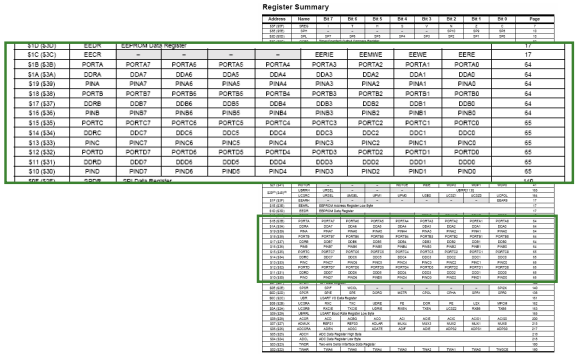
Port A Data Direction Register – DDRA

Bit	7	6	5	4	3	2	1	0	
	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	DDRA
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	

Port A Input Pins Address – PINA

Bit	7	6	5	4	3	2	1	0	
	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	PINA
Read/Write	R	R	R	R	R	R	R	R	
Initial Value	N/A	N/A	N/A	N/A	N/A	N/A	N/A	N/A	

Full I/O Register Map (pg. 329)



Register Summary

Address	Name	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0	Page
$1F ($5F)	SREG	I	T	H	S	V	N	Z	C	9
$1E ($5E)	EIND	–	–	–	–	–	–	–	EIND0	11
$1D ($5D)	SPL	SP7	SP6	SP5	SP4	SP3	SP2	SP1	SP0	11
$1C ($5C)	EECR	–	–	–	–	EERIE	EEMWE	EEWE	EERE	17
$1B ($5B)	PORTA	PORTA7	PORTA6	PORTA5	PORTA4	PORTA3	PORTA2	PORTA1	PORTA0	64
$1A ($5A)	DDRA	DDA7	DDA6	DDA5	DDA4	DDA3	DDA2	DDA1	DDA0	64
$19 ($59)	PINA	PINA7	PINA6	PINA5	PINA4	PINA3	PINA2	PINA1	PINA0	64
$18 ($58)	PORTB	PORTB7	PORTB6	PORTB5	PORTB4	PORTB3	PORTB2	PORTB1	PORTB0	64
$17 ($57)	DDRB	DDB7	DDB6	DDB5	DDB4	DDB3	DDB2	DDB1	DDB0	64
$16 ($56)	PINB	PINB7	PINB6	PINB5	PINB4	PINB3	PINB2	PINB1	PINB0	64
$15 ($55)	PORTC	PORTC7	PORTC6	PORTC5	PORTC4	PORTC3	PORTC2	PORTC1	PORTC0	65
$14 ($54)	DDRC	DDC7	DDC6	DDC5	DDC4	DDC3	DDC2	DDC1	DDC0	65
$13 ($53)	PINC	PINC7	PINC6	PINC5	PINC4	PINC3	PINC2	PINC1	PINC0	65
$12 ($52)	PORTD	PORTD7	PORTD6	PORTD5	PORTD4	PORTD3	PORTD2	PORTD1	PORTD0	65
$11 ($51)	DDRD	DDD7	DDD6	DDD5	DDD4	DDD3	DDD2	DDD1	DDD0	65
$10 ($50)	PIND	PIND7	PIND6	PIND5	PIND4	PIND3	PIND2	PIND1	PIND0	65
$0F ($4F)	SPDR	SPI Data Register								127

Instruction Classes (pg. 331)

- Arithmetic/Logic Instructions
- Data Transfer Instructions
- Program Control Instructions
- Bit Set/Test Instructions

Arithmetic/Logical Instructions

- Add Instructions
  - "ADD" Add Two Registers
  - "ADC" Add Two Registers and Carry
  - "INC" Increment a Register
  - "ADIW" Add Immediate to Word *
- Subtract Instructions
  - "SUB" Subtract Two Registers
  - "SBC" Subtract with Carry Two Registers
  - "SUBI" Subtract Immediate from Register*
  - "SBCI" Subtract with Carry Immediate from Register*
  - "DEC" Decrement Register
  - "SBIW " Subtract Immediate From Word**
- Compare Instructions
  - "CP" Compare Two Registers
  - "CPC" Compare with Carry Two Registers
  - "CPI" Compare Register and Immediate*
  - "CPSE" Compare Two Registers and Skip Next Instruction if Equal

16-bit and 32-bit support

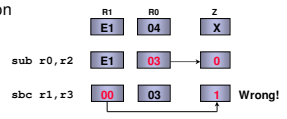
- Carry instructions
  - Addition, subtraction and comparison
  - Register with register or immediate
  - Zero flag propagation

```
SUB  R16,R24        SUBI  R16,1
SBC  R17,R25        SBCI  R17,0
```

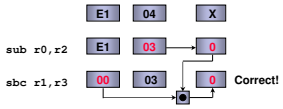
- All branches can be made based on last result
- Direct 16 bit instructions
  - Addition and subtraction of small immediates
  - Pointer arithmetics

Subtracting Two 16-Bit Values

- R1:R0 – R3:R2 (e.g., $E104 – $E101)
  - Without zero-flag propagation



  - With zero-flag propagation



Comparing Two 32-Bit Values

- Example: Compare R3:R2:R1:R0 and R7:R6:R5:R4

```
cp   r0,r4
cpc r1,r5
cpc r2,r6
cpc r3,r7
```

- After last instruction, status register indicates equal, higher, lower, greater (signed), or less than (signed)

Arithmetic/Logical Instructions (cont'd)

- Multiply instructions
  - "MUL" 8x8 –> 16 (UxU)
  - "MULS" 8x8 –> 16 (SxS)*
  - "MULSU" 8x8 –> 16 (SxU)**

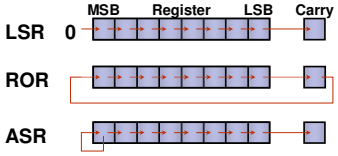
** Works on Registers R16 - R31*
*** Works on Registers R16-R23*

The result is present in R1:R0. All multiplication instructions are 2 cycles.

- Logical Instructions
  - "AND" Logical AND Two Registers
  - "ANDI" Logical AND Immediate and Register *
  - "OR" Logical OR Two Registers
  - "ORI" Logical OR Immediate and Register *
  - "EOR" Logical XOR Two Registers

Arithmetic/Logical Instructions (cont'd)

- Shift / Rotate Instructions
  - "LSL" Logical Shift Left
  - "LSR" Logical Shift Right
  - "ROL" Rotate Left Through Carry
  - "ROR" Rotate Right Through Carry
  - "ASR" Arithmetic Shift Right



Data Transfer Instruction Types

	# of Cycles
Data SRAM <–> Register File	(2)
Program Memory –> Register File	(1/3)
I/O Memory <–> Register File	(1)
Register File <–> Register File	(1)

Data Transfer RF <–> SRAM Stack

- "PUSH" PUSH a register on the stack
  - Decrements stack pointer by 1
  - Decremented by 2 when a return address is pushed on the stack
- "POP" POP a register from the stack
  - Increments stack pointer by 1
  - Incremented by 2 when a return address is popped off on return
- Stack grows from higher to lower memory locations

Flow Control

- Unconditional Jumps
- Conditional Branches
- Subroutine Call and Returns

Unconditional Jump Instructions

- "RJMP" Relative Jump *
- "JMP" Absolute Jump **

* *Reaches ± 2K instructions from current program location. Reaches all locations for devices up to 8KBytes (wrapping)*
** *4-Byte Instruction*

Conditional Branches (Flag Set)

- "BREQ" Branch if Equal
- "BRSH" Branch if Same or Higher
- "BRGE" Branch if Greater or Equal (Signed)
- "BRHS" Branch if Half Carry Set
- "BRCS" Branch if Carry Set
- "BRMI" Branch if Minus
- "BRVS" Branch if Overflow Flag Set
- "BRTS" Branch if T Flag Set
- "BRIE" Branch if Interrupt Enabled

Subroutine Call and Return

- "RCALL" Relative Subroutine Call *
- "CALL" Absolute Subroutine Call **

- "RET" Return from Subroutine
- "RETI" Return from Interrupt Routine

* *Reaches ± 2K instructions from current program location. Reaches all locations for devices up to 8KBytes (wrapping)*
** *4-Byte Instruction*

Bit Set/Clear and Bit Test Instructions

- "SBR" Set Bit(s) in Register *
- "SBI" Set Bit in I/O Register **
- "SBRS" Skip if Bit in Register Set
- "SBIS" Skip if Bit in I/O Register Set **
- "CBR" Clear Bit(s) in Register *
- "CBI" Clear Bit in I/O Register **
- "SBRC" Skip if Bit in Register Clear
- "SBIC" Skip if Bit in I/O Register Clear **

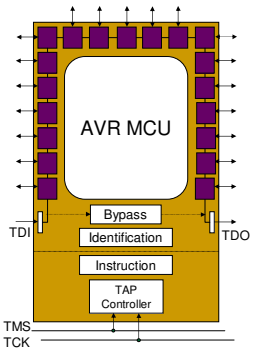
* *Works on Registers R16 - R31*
** *Works on I/O Addresses $00 - $1F*

Programming the ATmega16

- Traditional in-system programming
  - In-system programmable FLASH, EEPROM, and lock bits
  - Programmable at all frequencies
  - Programmable at all VCCs above 2.7V
  - Only four pins + ground required
  - Requires adapter device to control programming pins
- Self programming
  - The AVR reprograms itself without any external components
  - Re-programmable through any communication interface
    - Does not have to be removed from board
    - Uses existing communication ports
  - Critical functions still operating
    - device is running during programming

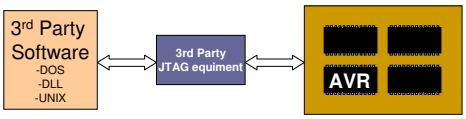
AVR JTAG Interface

- Complies to IEEE std 1149.1 (JTAG)
- Boundary-scan for efficient PCB test
  - Standard for interconnection test
  - All I/O pins controllable and observable from tester
- On-chip debugging in production



JTAG In System Programming

- The JTAG interface can be used to program the Flash and EEPROM
- Save time and production cost
  - No additional programming stage
  - Programming time independent of system clock



JTAG In-Circuit Emulator

- Controlled by AVR Studio
- Real-Time emulation in actual silicon
  - Debug the real device at the target board
  - Talks directly to the device through the 4-pin JTAG interface
- Supports
  - Program and Data breakpoints
  - Full execution control
  - Full I/O-view and watches

AVR Studio

- Integrated development environment for AVR
- Front end for the AVR simulator and emulators
- C and assembly source level debugging
- Supports third party compilers
- Maintains project information
- Freely available from www.atmel.com
- Third-party compilers