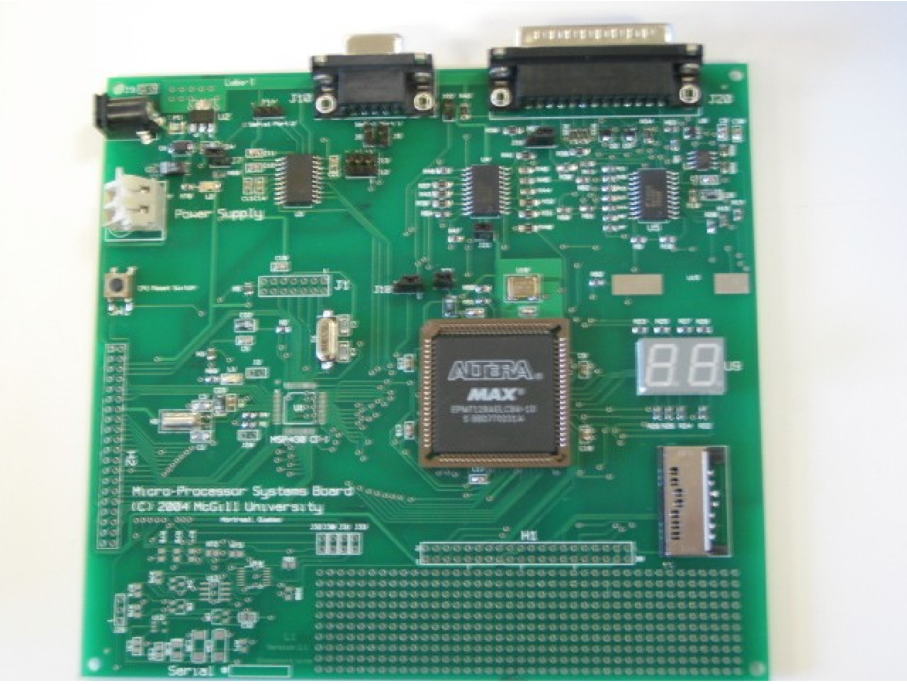
ECSE-426

Microprocessor Systems



Today's Lecture

- Theory
  - Multi-level machines
  - Problem-oriented language layer
  - Language Choice
  - Machine Architecture
- Tutorial
  - Introduction to the MSP430
  - Registers
  - Clocks and Timers
- Appendix
  - Report Guidelines

Contemporary Multilevel machines

5	Problem-oriented Language Level	
		Translation (Compiler)
4	Assembly Language Level	
		Translation (Assembler)
3	Operating System Machine Level	
		OS - Partial Interpretation
2	Instruction Set Architecture Level	
		Interpreter
1	Microarchitecture Level	
		Hardware
0	Digital Logic level	

Problem-oriented Language layer

- Compiled to assembly or instruction set level
- You will be using embedded C
- How does this differ from usual use of C?
  - Directly write to registers to control the operation of the processor
  - All of the MSP430 registers have been mapped to macros
  - Important bit combinations have macros – use these, please !
  - Registers are 16 bits, so int type is 2 bytes
  - Register values may change without your specific instructions
  - Limited output system
  - Floating point operations very inefficient, divide + square-root to be avoided.

Assembly versus C

- Efficiency of compiled code
- Source code portability
- Program maintainability
- Typical bug rates (say, per thousand lines of code)
- The amount of time it will take to develop the solution
- Availability and cost of compilers and other development tools
- Your personal experience (or that of the developers on your team) with specific languages or tools

- Don’t rule out Java or C++ if you have the memory to play with.

Problem

- Company “Ostrich” has recently re-developed their embedded software for flagship products
  - Developed in assembly, 80 percent working, 2000 lines of code
- Suddenly realized that the product is far from shippable
- Bugs: system lock-ups indicative of major design flaws or implementation errors + major product performance issues
- Designer has left the company and provided few notes or comments

- You are hired as a consultant. Do you:
  - Fix existing code?
  - Perform complete software redesign and implementation? In this case, which language?

Points

- Fix existing firmware:
  - Major risk: may not even be possible
- Complete Redesign
  - Definitely work, but may take longer

"I study the market research, analyze any changes in our customer base, then drink gallons of coffee until I hallucinate a big pink rabbit who tells me what to do."

- Languages
  - Assembly: rarely the right choice. Suffers from lack of portability and maintainability. Here you are faced with an unfamiliar assembly language
  - C: small microprocessor, 8K of ROM and 368 bytes RAM

Approach

- Fix one of the minor bugs and see how long it takes
  - 22 hours
  - Reveals that original author was not proficient in the assembly language and had no clear master-plan
- Estimation of how many hours you will need for new design: 2 months
- Language: prefer C, but need to check
  - No size or efficiency surprises
- What do you need beyond assembly?
  - Extra RAM usage for frequent function calls
  - Extra ROM usage: C startup code and compiler-supplied routines (e.g., no multiplier instruction)
  - Any performance impact due to compiler usage

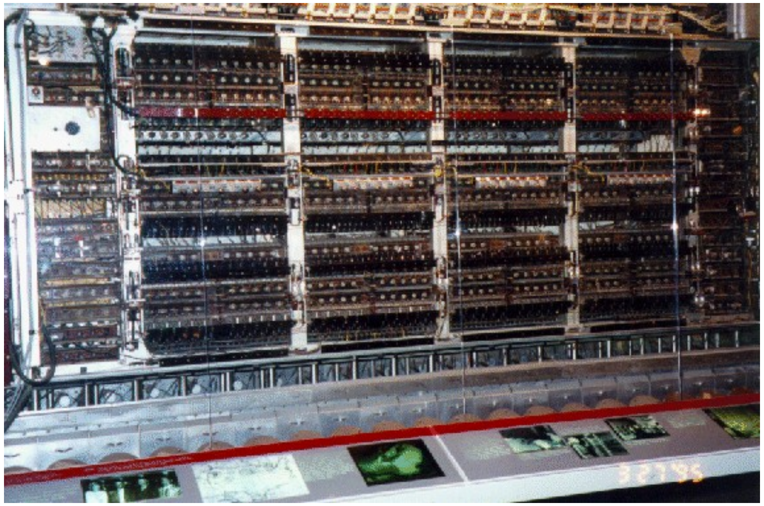
Findings

- Compiler experimentation
  - overlays to minimize RAM usage
  - C startup code was very small
  - Multiply instruction adds a few hundred lines of code
  - Existing code: 2K instructions, 4K look-up tables
- The Test
  - Fix existing code: 86 hours (no bugs, no performance improvement)
  - Rewrite in C: 185 hours

  - C implementation has design and performance improvements
  - Maintainable and portable code
  - Modular implementation, carefully commented, passed through a code inspection.
  - C code: faster than original assembly, smaller image.

Von Neumann Machine

- Princeton IAS



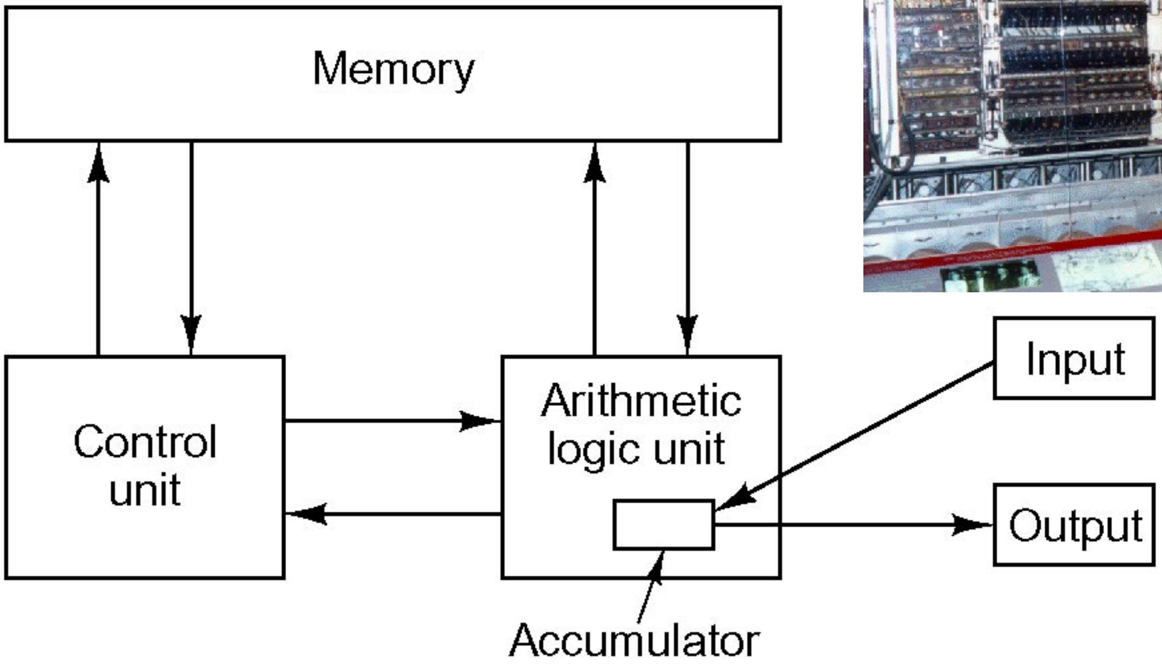
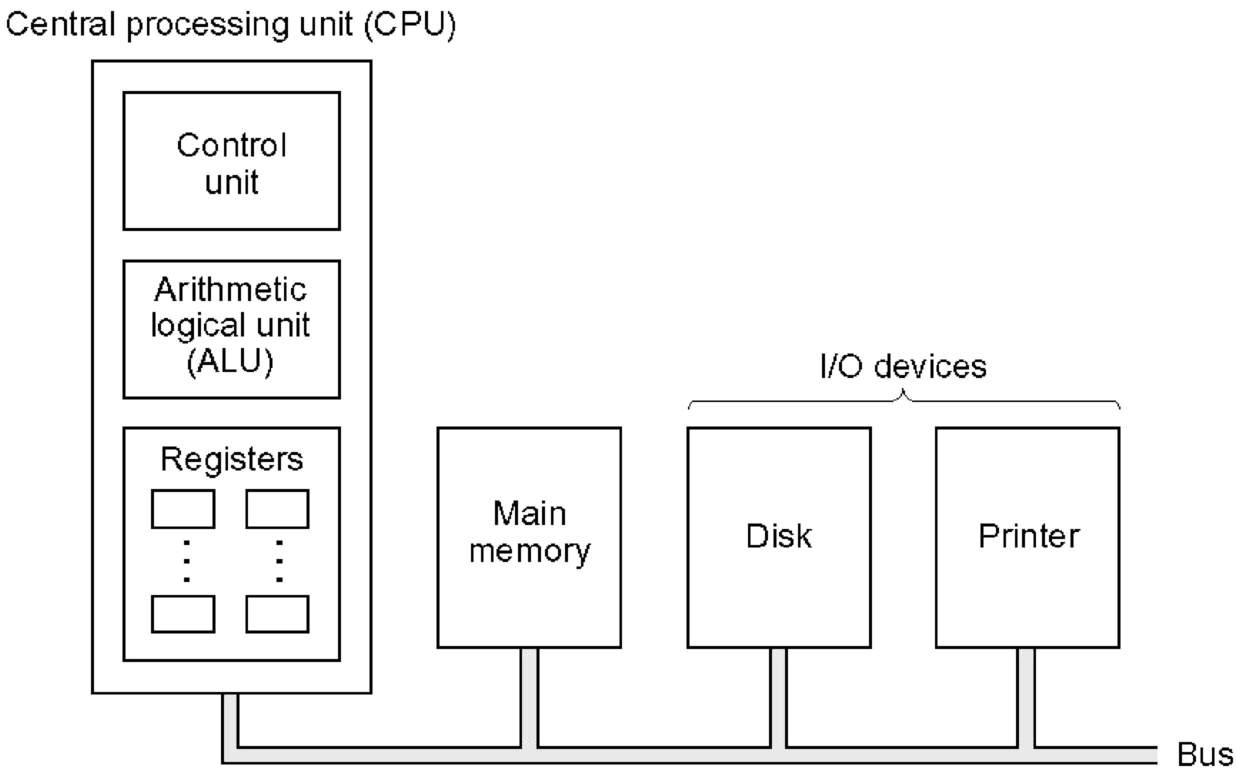


*Image from : http://wwwcsif.cs.ucdavis.edu/~csclub/museum/items/ias.html

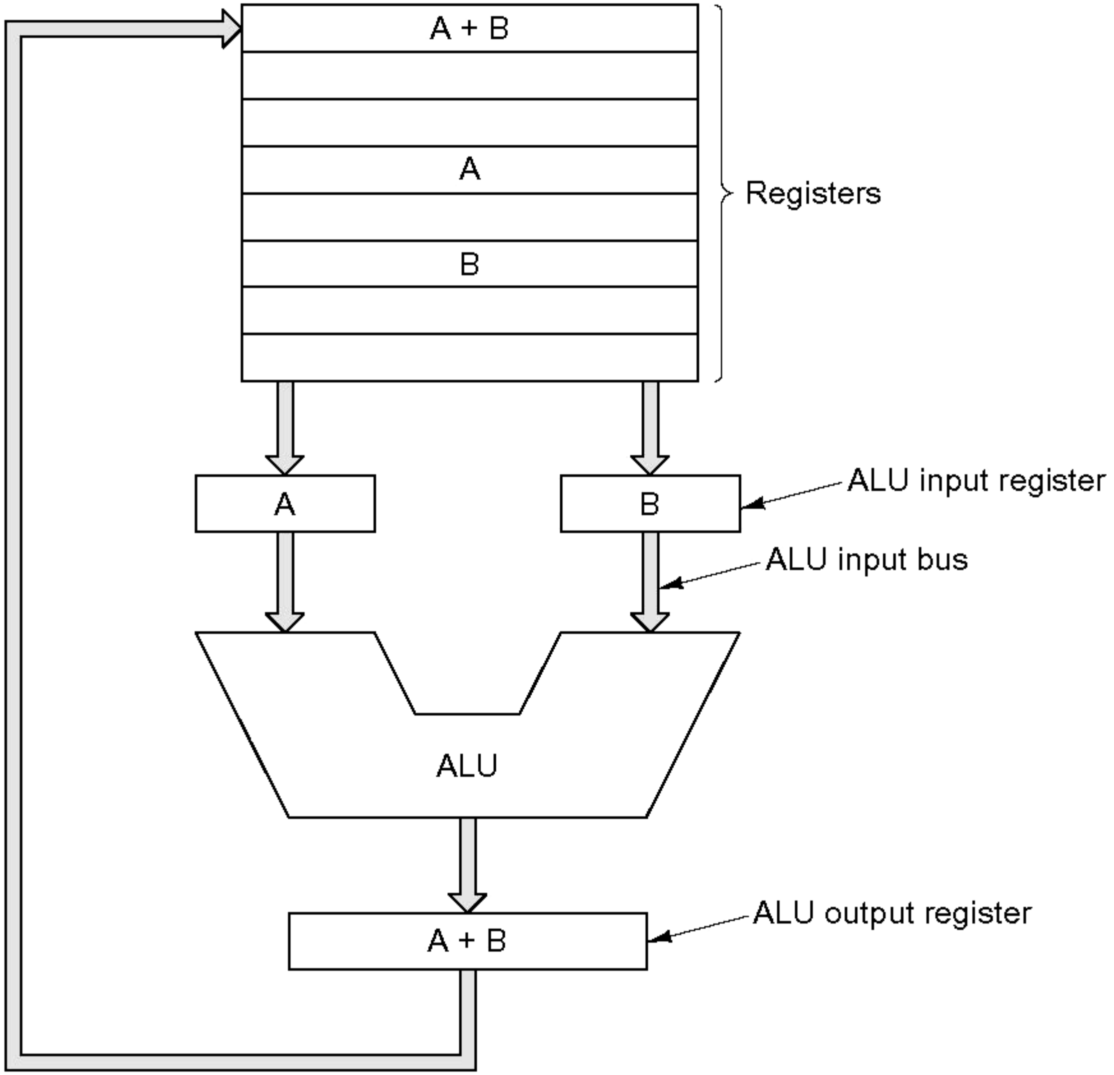
Computer Organization

- Processor
  - Microprocessor
- Memory
- Peripherals
- Common Bus



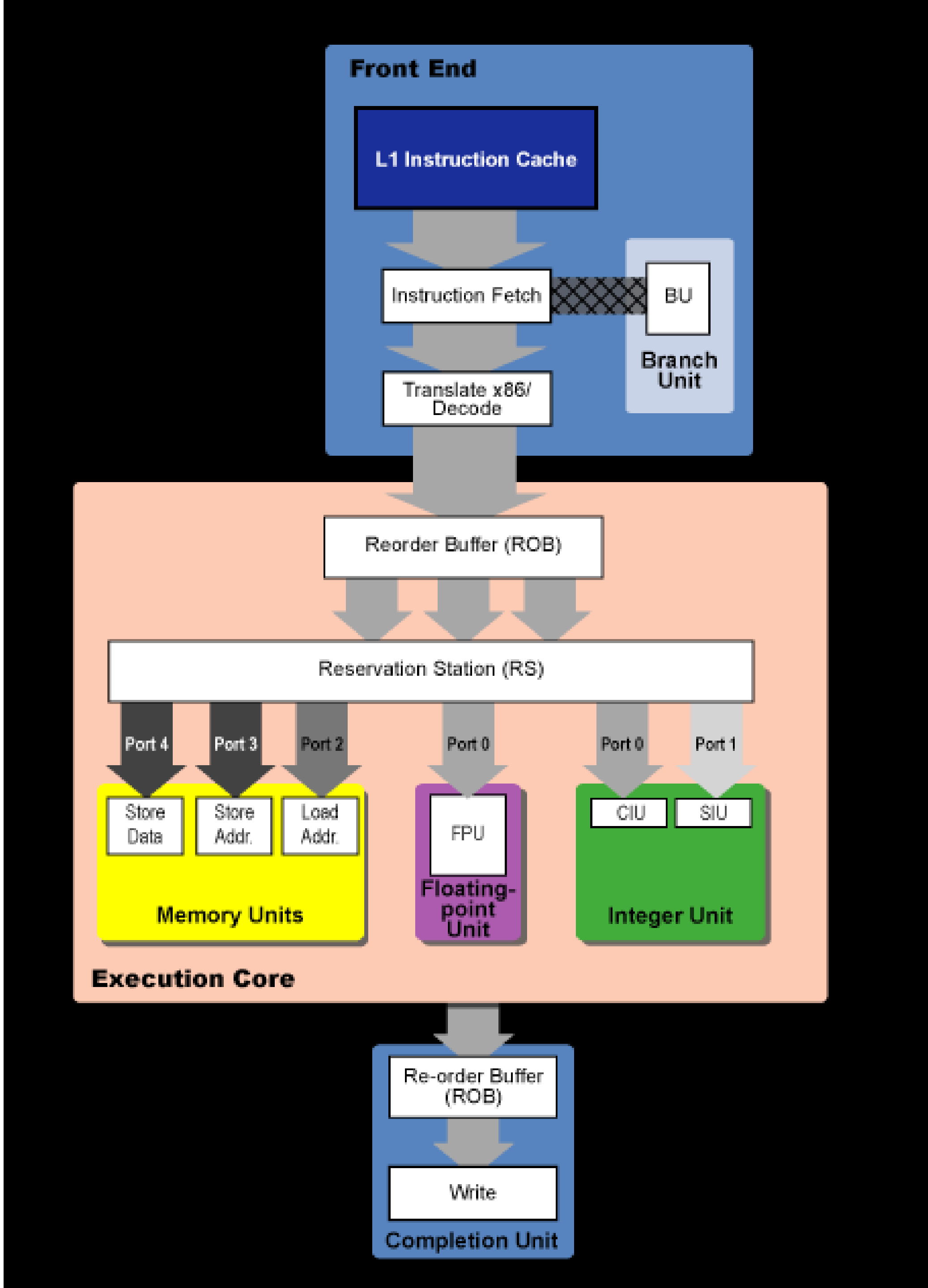
Common Processors

- Still Von Neumann architecture
  - Arithmetic-logic unit
  - Registers
  - Auxiliary registers

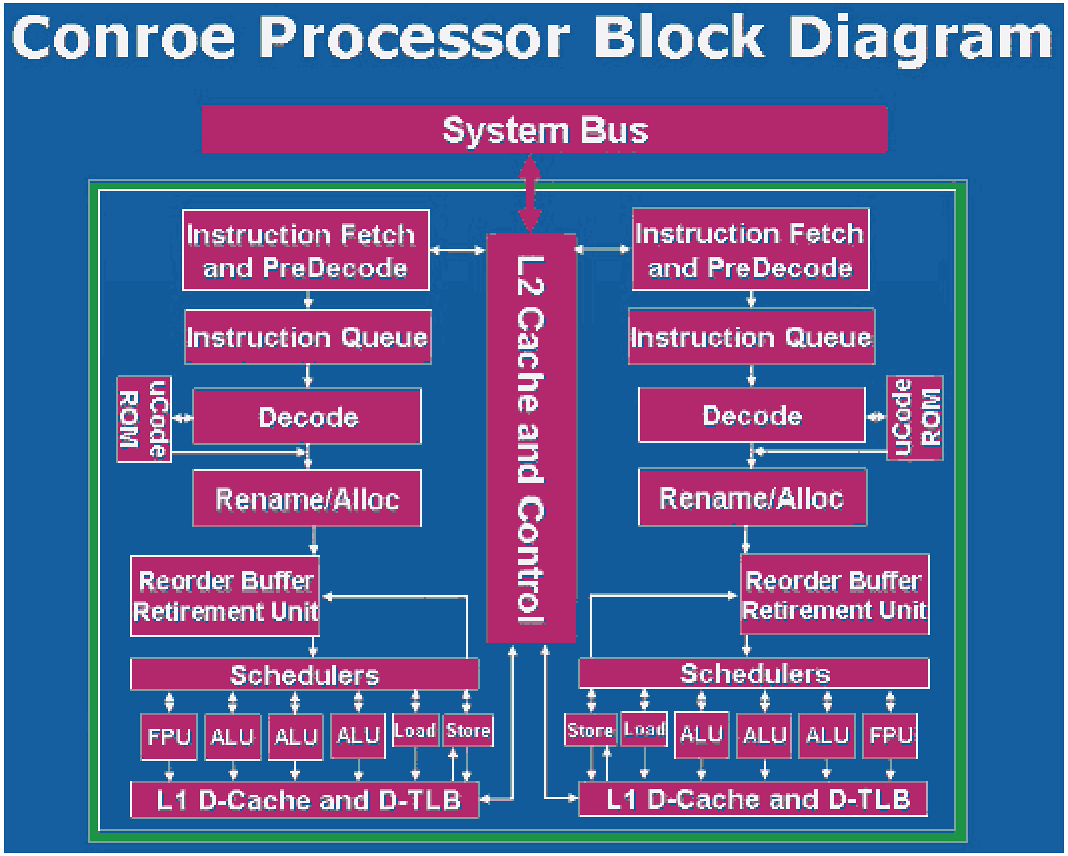


Some complications…

- Pentium 6 architecture



Dual-core architectures

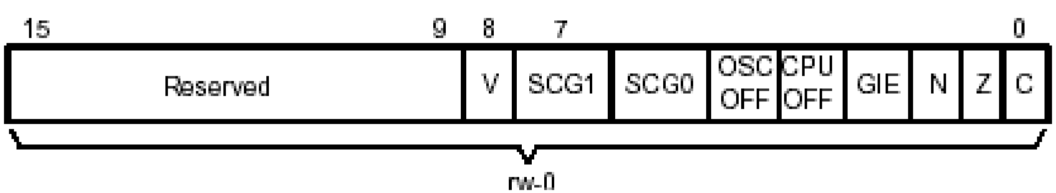


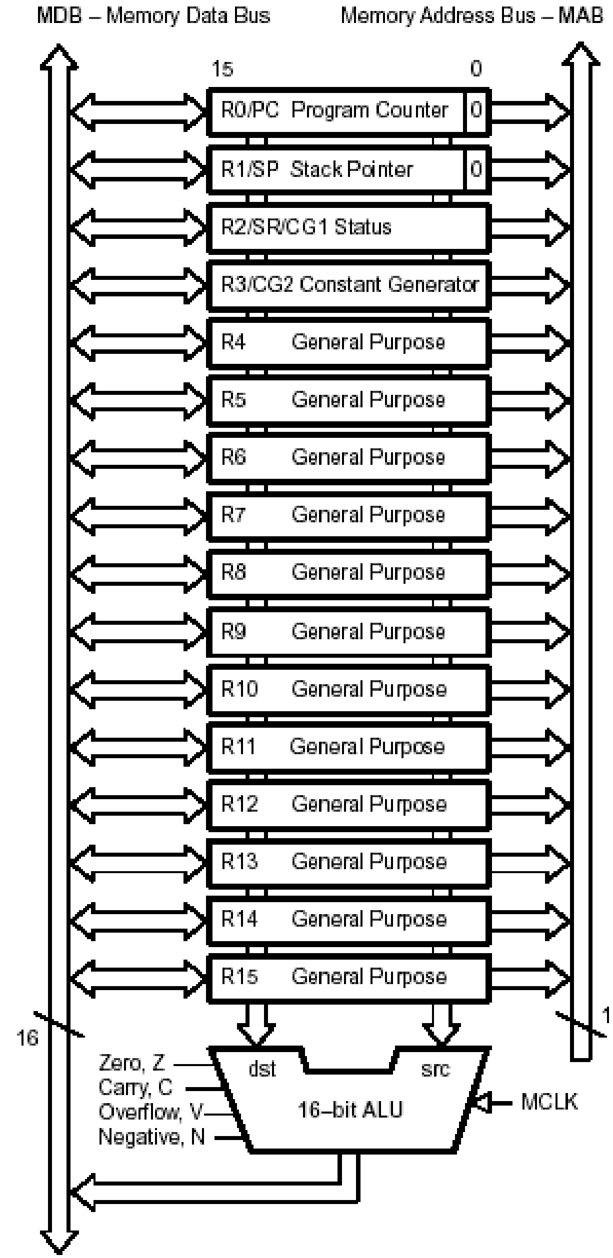
Harvard Architecture

- Separate Instructions from Data on two Signal Paths
  - Full control of instruction bus size (no need to be multiples of 8 bits.
  - Can fetch data and instructions simultaneously
    - Useful for DSP applications
- Modern architectures implement a hybrid of both
  - Von Newmann at the highest level
  - Harvard architecture (modified) as a consequence of using instruction and data caches
- PIC and AVR microcontrollers are examples of Harvard architectures.

MSP430 ISA

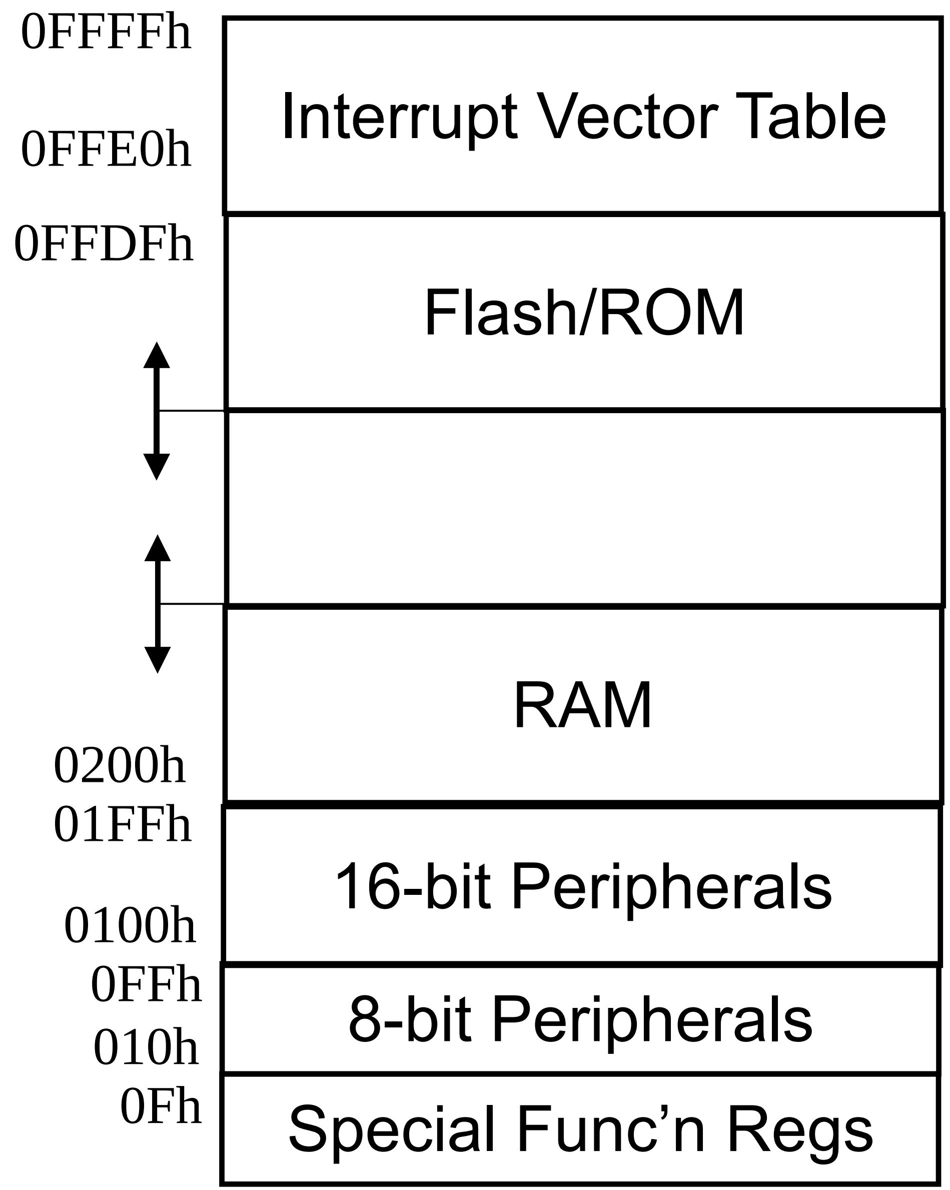
- A 16-bit RISC processor
- Small set of 27 instructions
- Large register set
  - General purpose registers
  - Program Counter, Stack Pointer
  - Constant Generator
- 16-bit memory access
  - PC, SP LSB bit set to 0
- Status Register





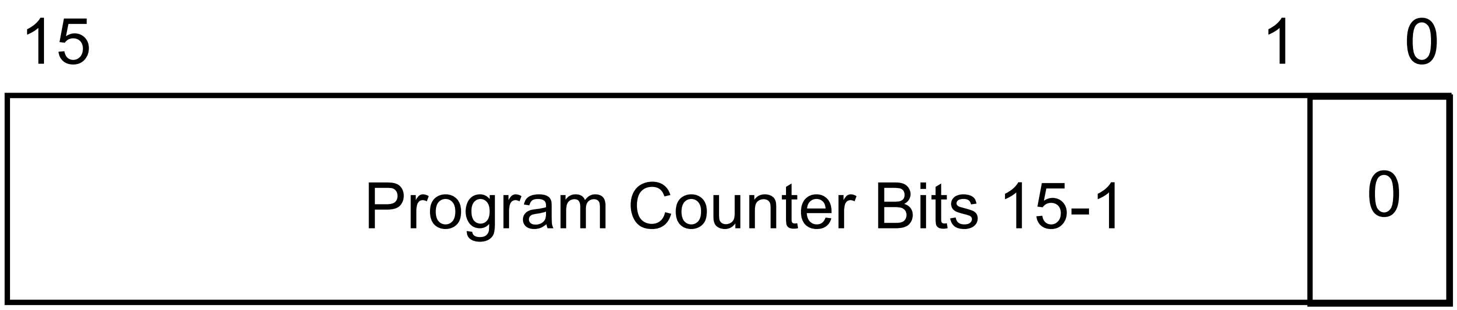
Address Space

- 60K ROM
- 2K RAM
- Special Function Registers (SFRs) accessed by byte instructions



Program Counter

- Tells you where you are in the program
- 16 bits, aligned to even addresses
  - MOV #LABEL, PC Branch to LABEL
  - MOV LABEL, PC Branch to address in LABEL
  - MOV @R14, PC Branch indirect to address in R14



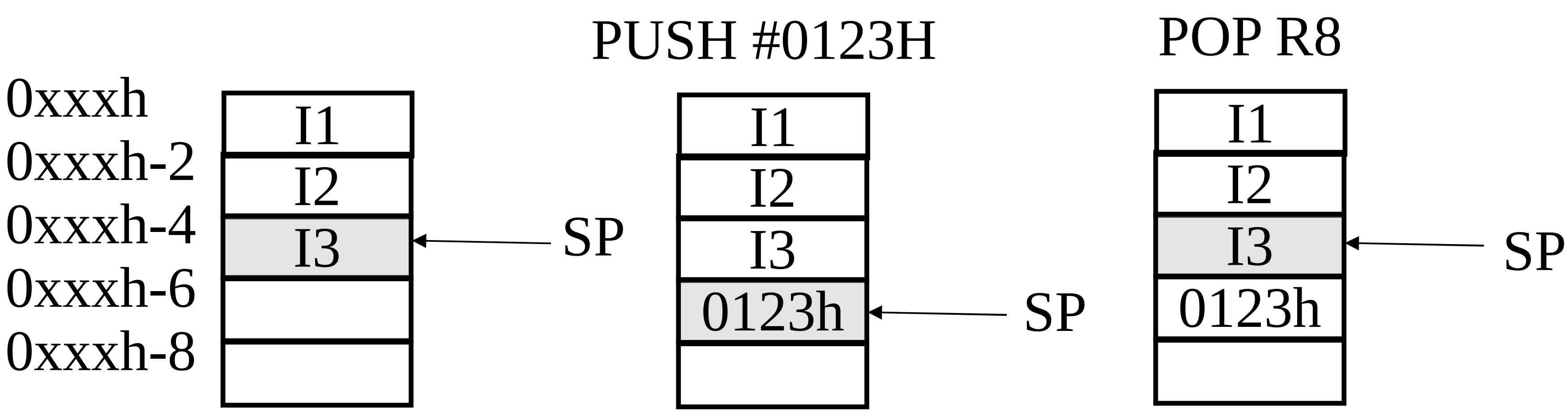
Stack Pointer

- Used by CPU to store return addresses of calls and interrupts
- Don’t push and pop the SP and PC !
- SP can also be used by software.
- Initialized into RAM by user, aligned to even addresses

15 … 1	0
Stack Pointer Bits 15-1	0

Stack Pointer Usage

- MOV 2(SP), R6 ; Item I2 -> R6
- MOV R7, 0(SP) ; Overwrite TOS with R7
- PUSH #0123H ; put 0123H onto stack
- POP R8 ; R8 = 0123H



Status Register

- V: overflow, N: negative, Z: zero, C: carry
- JEQ, JN, JZ, JC
- SCG0/SCG1 : Turn off SMCLK, DCO
- OSC OFF, CPU OFF: turn off Oscillator or CPU
- GIE: general interrupt enable (enable maskable interrupts)

15 9	8	7					1	0
RES'd	V	SCG1	SCG0	OSC OFF	CPU OFF	N	Z	C

Constant Generator Registers

- R2 and R3 can generate constants
- Source operand only
- Addressing mode determines the returned constant.
- Available: 0,1,2,4,8, -1 (=FFFFh)
- Advantages:
  - No special instructions
  - No additional code words
  - No code memory access

Watchdog Timer

- Hardware
  - used to automatically detect software anomalies
  - resets the processor if any occur.
- Implemented as a counter (counts down to zero)
- The embedded software selects the counter's initial value and periodically restarts it.

- If counter ever reaches zero before software-restart
  - software is presumed to be malfunctioning.
  - processor's reset signal is asserted.
  - processor (and executing software) will be restarted as if a human operator had cycled the power.

Watchdog Timer

- Timer: perform controlled system restart after software problem occurs (32 ms reset interval)
- Watchdog mode or interval timer mode.

- If you debug, you need to stop the watchdog:

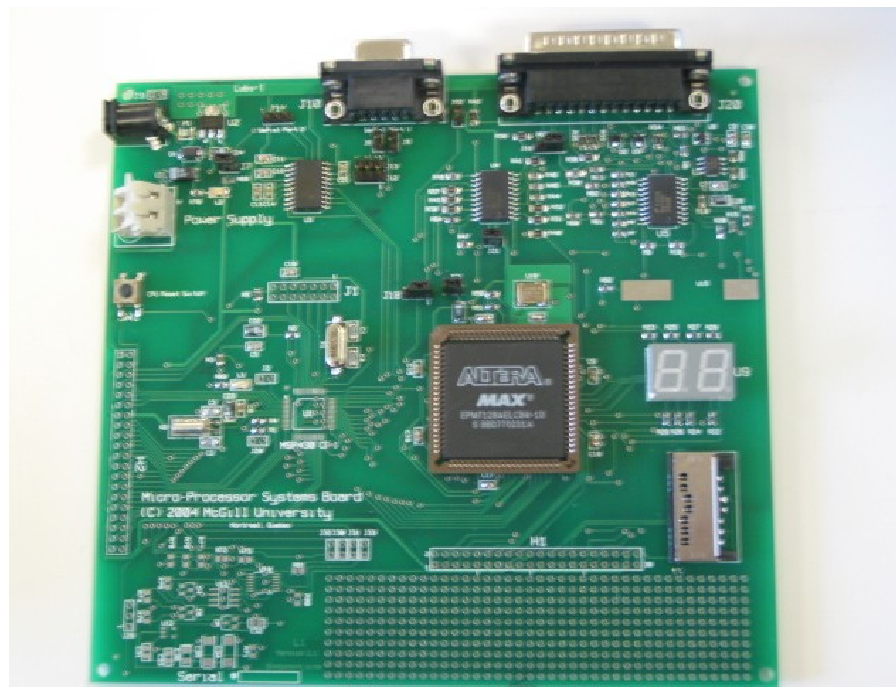
- mov.w #WDTPW + WDTHOLD, &WDTCTL

- WDTPW : Watchdog Timer Password
- WDTHOLD: Watchdog Timer Hold

- WDTCTL: Watchdog Timer Control Register

Microp Board Overview

- Processor: MSP430F149
- Programmable Logic Device MAX7128AE
- Power supply +5V DC or through data input
- Parallel and Serial (RS232 C) interface
- Smart Card Media socket
- Hex (LCD) Display
- iButton clip
- 40-pin expansion connector
- Prototyping area
- [Reference: Schematics]



MicroP board components

- RSR232 interface (Serial Port) (U3)
- Smart Card Media socket (U11)
- Hex Display (U9)
- iButton clip (U15)
- 8MHz crystal (X1) and 32.768KHz crystal (X2)
- 40MHz oscillator (U10)
- MSP430F149 MCU (U1)

Board Layout

- Each board has a serial number on the bottom (viewed from the component side) left of the board.
- Components, headers, resistors and capacitors, and oscillators are labeled “U”, “H”, “R”, “C”, “X” respectively.
- Each character is followed by a number indicating the unique component.
- For example, U8 refers to the MAX7128AE CPLD.

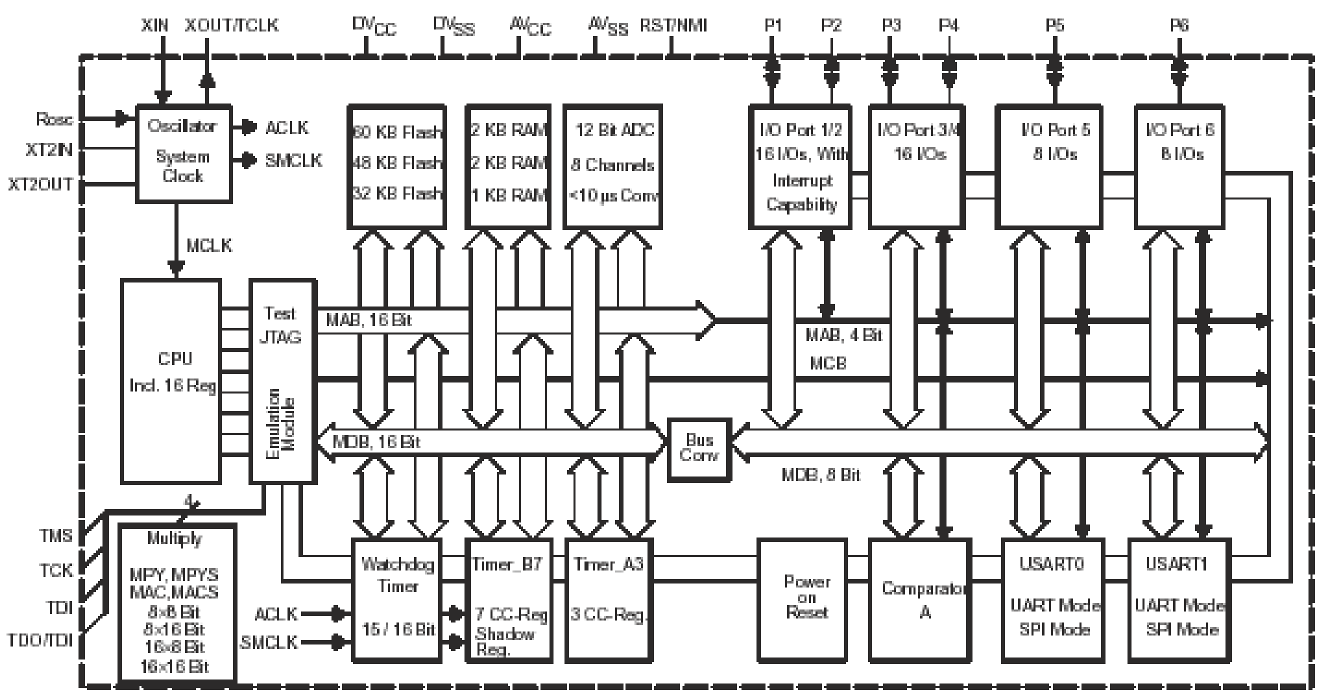
Board Components

- MAX7128AE PLD (U8)
- Serial communication interface (J10)
- Parallel port interface (J20)
- Headers
  - 2x10 headers (H1) for connection to PLD and HEX display
  - Shrouded straight 40-pin header (H2) for connection to a daughter card
- 1.1 User Manual Notation

MPS430 Block Diagram

- Several families
- Small ISA, large set of peripherals
  - Flash, USART, Timers, AD/DA, GPIO
- 3xx
  - Basic
- 1xx
  - Rich
- 4xx
  - LCD
- Our model MSP430F149

MSP430x14x



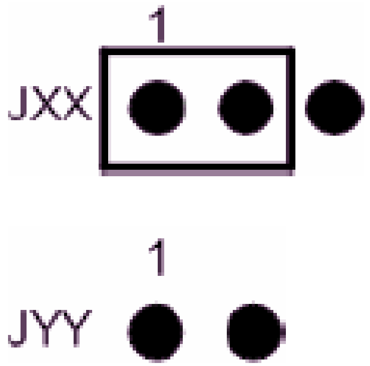
MSP430x Peripherals - IO Pins

- General-purpose and specialized IO
- GPIO Ports
  - Input/Output
  - Interrupt generation (option)

	Port 0 ('3xx only)	Port 1	Port 2	Port 3	Port 4	Port 5	Port 6
Input	P0IN 010h	P1IN 020h	P2IN 028h	P3IN 018h	P4IN 01Ch	P5IN 030h	P6IN 034h
Output	P0OUT 011h	P1OUT 021h	P2OUT 029h	P3OUT 019h	P4OUT 01Dh	P5OUT 031h	P6OUT 035h
Direction	P0DIR 012h	P1DIR 022h	P2DIR 02Ah	P3DIR 01Ah	P4DIR 01Eh	P5DIR 032h	P6DIR 036h
Interrupt Flags	P0IFG 013h	P1IFG 023h	P2IFG 02Bh	Not Implemented	Not Implemented	Not Implemented	Not Implemented
Interrupt Edge Sel	P0IES 014h	P1IES 024h	P2IES 02Ch	Not Implemented	Not Implemented	Not Implemented	Not Implemented
Interrupt Enable	P0IE 015h	P1IE 025h	P2IE 02Dh	Not Implemented	Not Implemented	Not Implemented	Not Implemented
Function Select	Not Implemented	P1SEL 026h	P2SEL 02Eh	P3SEL 01Bh	P4SEL 01Fh	P5SEL 033h	P6SEL 037h

Jumpers

- Jumpers are labelled “J”, followed by a number.
- Pin placements
- (a) Jumper block on pin 2 and pin 3 of a 3-pin jumper JXX
- (b) Open connection – no jumper blocks required (JYY)
- Note that pin 1 on the jumpers is explicitly labelled and has a square connector on the board.



Power

- Variety of power supplies.
  - The power supply (except when powering from parallel port) is regulated through a 3.3V regulator and a reset-able 500mA fuse.
- Parasitic power
  - Can operate from a regulated 1.8V to 3.6V supply.
  - For low power applications, parallel port supplies 2.8V.
  - This is not be enough to power for using the CPLD.
  - When using power from the parallel port, remember to remove the CPLD power supply from the board.
  - The RS-232C serial interface should also be unplugged.
  - Struggle from a Laptop!

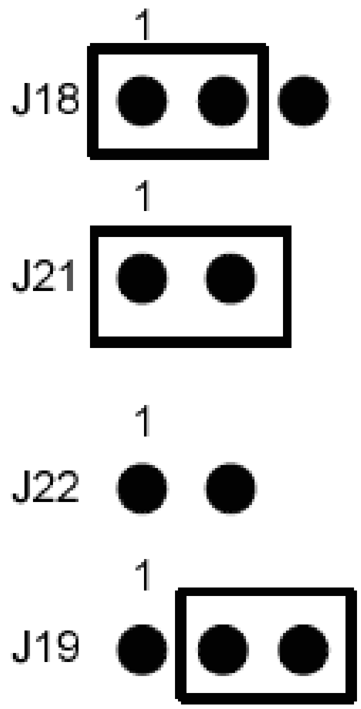
Power

- External Wall-mount Supply
  - With the use of the 78M33C regulator, MCGUMPS can operate from an external wall supply. The power connector is a 2.5mm barrel connector.
- 2-input connector terminal block
  - Minimum Vin = 4.3V is recommended for the 3.3V regulator.
  - Supplied from the standard DC lab power supply.
  - Voltage on the input power line should not exceed 10V because of user safety concerns and the protection of some of the 3.3V digital lines.

Programming the MSP430

- The MSP430 can be programmed through JTAG chain using Altera/Quartus and also through CrossWorks.

- Jumper settings for MCU via Crossworks and CPLD via Altera/Quartus:



Oscillators

- 40 MHz (U10) – Pin 83 of CPLD
- 8 MHz (X1) – Pins 52 and 53 of MSP430
- 32.768 KHz (X2) – Pins 8 and 9 of MSP430

Break ?

- 5 minutes
- Next:
  - Programming the MSP430
  - MSP430 Clocks, Timers
  - MSP430 Interrupts
  - Embedded C Tips
  - Guidelines for experiment reports

Programming the MSP430

- Startup
  - Configure McGumps if required.
  - Power up the board.
  - Connect the board to the terminal.
  - Apply power to the board.
- Resetting and Aborting Programs
  - The MCU can be reset by pressing the reset button (SW1).
  - This operation has the effect of restarting the program currently loaded on the MCU.
  - If the program running on the MCU has to be aborted, the board has to be powered down by removing the power connection.

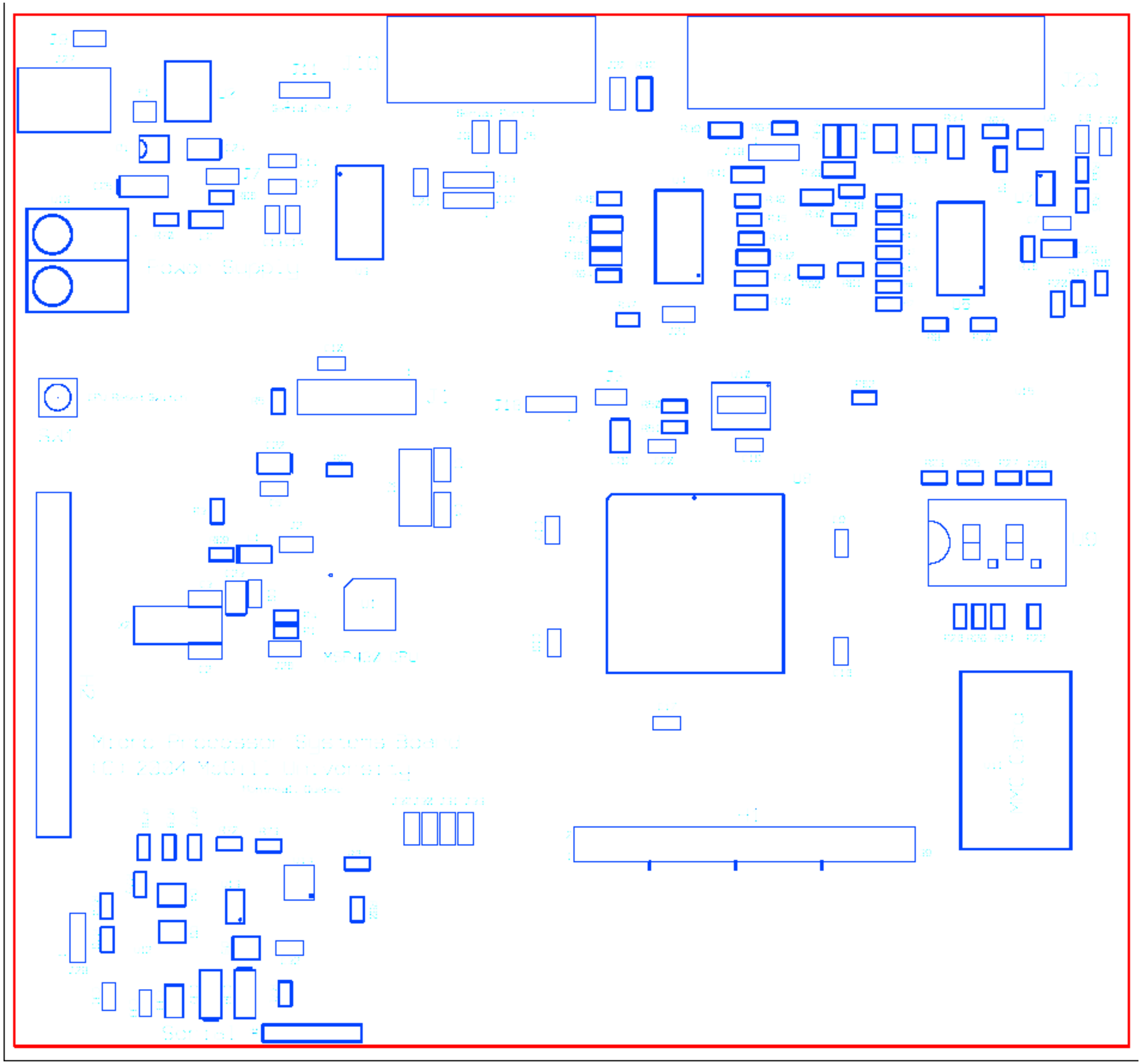
Flashing LED Example

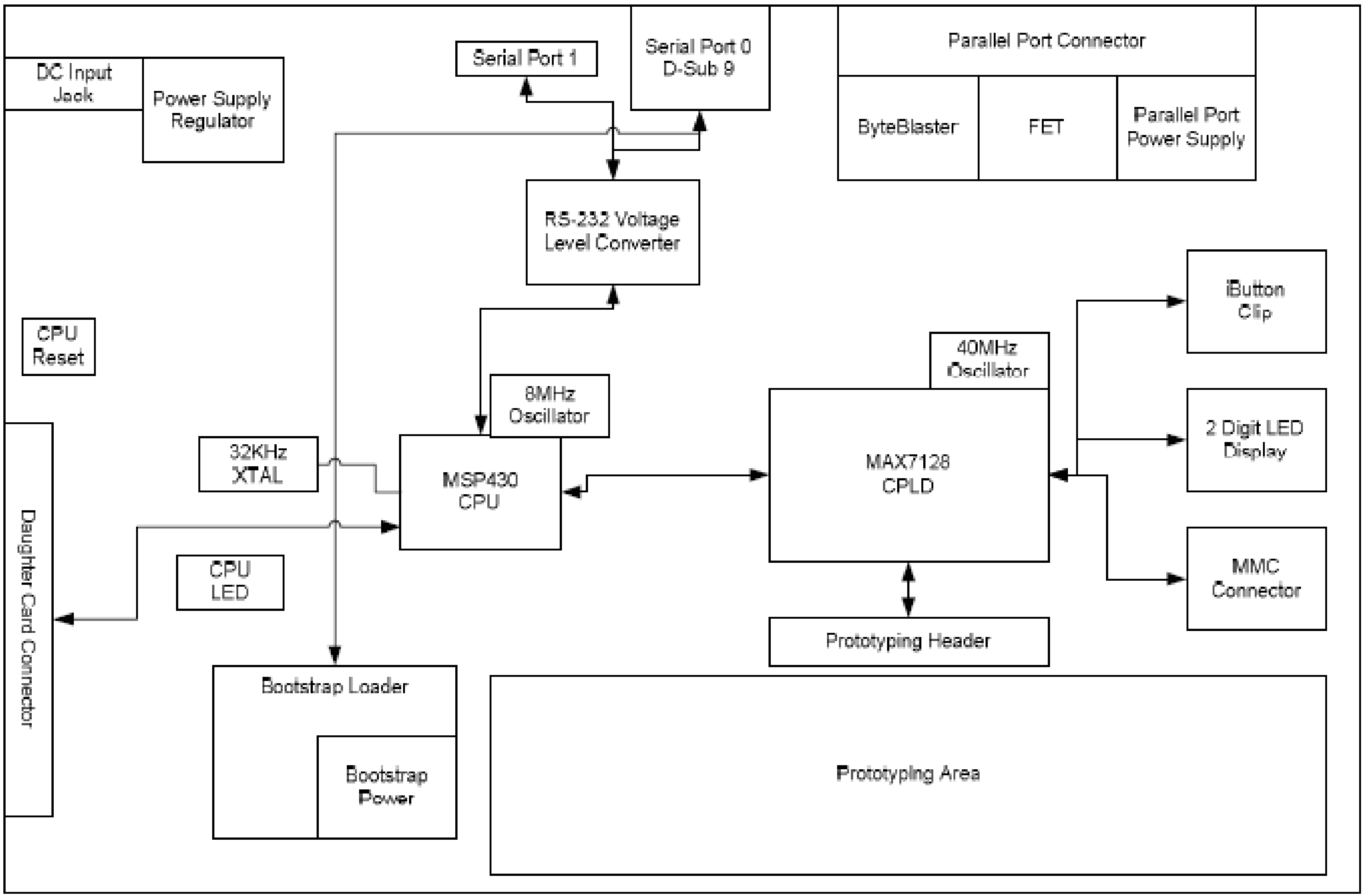
1) Start (Rowley Associates Limited) CrossWorks MSP430 1.2. and create new solution.
2) In New Project window,
    1) choose TI FET Projects folder under Project Types
    2) choose FET430P 140 C Project under templates
3) A demo that flashes the LED is shown in the main.c tab in the main frame.
4) To compile, Build -> Build Solution. In the Output frame, the build log indicates that build is complete.

Flashing LED Example (ctd)

1) Ensure that the jumpers are set-up to enable programming of the MSP430 MCU.
2) Ensure that the parallel port 1 is connected to the board. Choose Target -> Connect FET on LPT1 to connect to the MSP430 microprocessor.
3) Choose Debug->Start Debugging to download flashing LED example on to MSP430 microprocessor. The LED should start flashing.

Note: In the above example, the energy needed for operation is taken from the parallel port (parasitic power).




Serial Port 1
Serial Port 0
D-Sub 9
Parallel Port Connector
DC Input
Jack
Power Supply
Regulator
ByteBlaster
FET
Parallel Port
Power Supply
RS-232 Voltage
Level Converter
iButton
Clip
CPU
Reset
40MHz
Oscillator
8MHz
Oscillator
2 Digit LED
Display
32KHz
XTAL
MSP430
CPU
MAX7128
CPLD
Daughter Card Connector
CPU
LED
MMC
Connector
Prototyping Header
Bootstrap Loader
Bootstrap
Power
Prototyping Area

MSP430 Clocks

- Three clock sources and three clock lines – allows a mix of slow and fast clocks in the system.

- Low Frequency Crystal Clock (LFXTCLK) –
  - Crystal connected to the XIN and XOUT pins with intended oscillation of 32kHz.
  - Always the source of the Auxiliary Clock line (ACLK).
  - This source can be turned off with the OSCOFF option in the Status Register.

- Crystal 2 Clock (XT2CLK) –
  - 8 MHz crystal connected to the XT2IN and XT2OUT pins.
  - In general, this signal is meant to be the high-speed clock source.
  - This source can be turned off with the XT2OFF bit of the Basic Clock system control register 1 (BCSCTL1).

MSP430 Clocks (ctd)

- Digitally Controlled Oscillator Clock (DCOCLK)
  - the only internally generated clock input
  - default clock source for the master clock upon reset.
  - By default this clock runs at about 900kHZ
  - The RSELx, MODx, and DCOx bits allow adjustment

MSP430 Clock Lines

- Master Clock (MCLK)
  - Source for the MSP CPU core;
  - Must be working properly for the processor to execute instructions.
  - Source is selected with the SELMx bits of the Basic Clock System Control Register 2 (BCSCTL2).
  - The divider is controlled with the DIVMx of the BCSCTL2.
  - CPU can be turned off with the CPUOFF bit of the Status Register (SR), but to recover from this state an interrupt must occur.
- # Submaster Clock (SMCLK) - This clock is the source for most peripherals, and its source can either be the DCO or Crystal 2. The source clock is controlled with the SELS and SCG bits of the BCSCTL2 and SR. The divider is controlled by the DIVSx bits of the BCSCTL2.

MSP430 Clock Lines (ctd)

- # Auxiliary Clock (ACLK) - this clock line's source is always LFXTCLK. It is an option for slower subsystems to use in order to conserve power. This clock can be divided as controlled by the DIVAx bits of the Basic Clock System Control Register 1 (BCSCTL1).

Timers

- Two digital timers, A & B.
- Timer A:
  - 3 capture/compare registers
  - Interrupts: via overflow, or from one of the c/c registers.
  - Selectable clock source
  - Configurable outputs with PWM (pulse-width modulated) capability
  - Interval timing

Timer A

- 16 bit timer/counter register, TAR
  - increments/decrements with each rising edge of clock
  - Software read/write
  - Interrupt on overflow.
  - Clear: TACLR bit (also clears clock divider and count direction)
- Clock source
  - ACLK, SMCLK, or externally (TACLK or INCLK)

Timer A

- Start timer
  - MCx > 0 and clock source active
  - In up or up/down mode
    - write 0 to TACCR0 to stop timer
    - then write nonzero value to TACCR0 to start.
- Modes
  - Stop: timer halted
  - Up: zero to TACCR0 repeatedly
  - Continuous: zero to 0xFFFFh
  - Up/down: zero to TACCRO, back to zero

Timer A Modes

- Up mode:
  - Used for periods different from 0xFFFFh
  - Timer counts per period is TACCR0+1
  - TACCR0 CCIFG interrupt flag set when timer counts to TACCR0 value
  - TAIFG flag when timer counts to zero
- Continuous mode
  - TAIFG flag set when timer counts from 0xFFFFh to 0

Timer A Modes

- Up/down mode:
  - Used for symmetrical pulse generation
  - Counts to TACCR0 and back to 0 (repeatedly)
  - TACCR0 CCIFG interrupt flag set when timer counts to TACCR0 value
  - TAIFG flag when timer counts to zero
  - Interrupt flags separated by ½ timer period
  - Supports applications that require dead times.

Capture/compare blocks

- Capture mode (CAP = 0)
  - Used to record time events (speed computations, time measurements)
  - Capture occurs on selected edge of input
    - Timer value copied to TACCRx register
    - Interrupt flag CCIFG is set
- Compare mode (CAP = 1)
  - Used to generate PWM output signals or interrupts at specific time intervals
  - When TAR counts to TACCRx
    - Set CCIFG, set internal signal EQUx = 1, latch CCI to SCCI.

Timer A Interrupts

- Two interrupt vectors
- TACCR0 vector for TACCR0 CCIFG
- TAIV vector for all other CCIFG flags and TAIFG.
  - In capture mode, a CCIFG flag is set when a time value is captured. In compare mode, a CCIFG flag is set if TAR counts to the associated TACCRx value.
- TACCR0 CCIFG : highest priority
  - Dedicated interrupt vector

Timer A Interrupts

- TAIV vector - prioritized
  1. TACCR1 CCIFG
  2. TACCR2 CCIFG
  3. TAIFG (overflow flag)

Interrupts

- Processing external events
  - Asynchronous to program execution
  - Mostly I/O driven, but also timers, SW exceptions
  - Different from traps (synchronous with program)
- Example of interrupt (like exp. 3)
  - Goal: output line of characters to terminal
  - Collect characters in buffer
  - Initialize a pointer (ptr) and a counter (count)
  - Check if terminal is ready and start I/O
  - After character is displayed -> interrupt

Interrupts (example cont'd)

- Hardware
  (1) Device controller asserts interrupt line on system bus
  (2) When ready, CPU asserts interrupt acknowledge signal
  (3) Device controller puts integer (interrupt vector) on data lines to identify itself
  (4) CPU removes the interrupt vector & saves it
  (5) CPU pushes PC and PSW onto stack
  (6) CPU locates new program counter using the interrupt vector as index into a table at the bottom of memory.

Interrupts (example cont'd)

- Software
  (1) Interrupt service routine saves all registers
  (2) Read device register to determine terminal number
  (3) Read status codes for the interrupt
  (4) Handle a potential I/O error
  (5) Increment ptr, decrement count. If count > 0, copy *ptr to output register
  (6) If required, output special code to tell device or interrupt controller that interrupt has been processed
  (7) Restore all saved registers.
  (8) Execute the return from interrupt (RETI) instruction, restorting state/mode of CPU.

Transparency & Priority

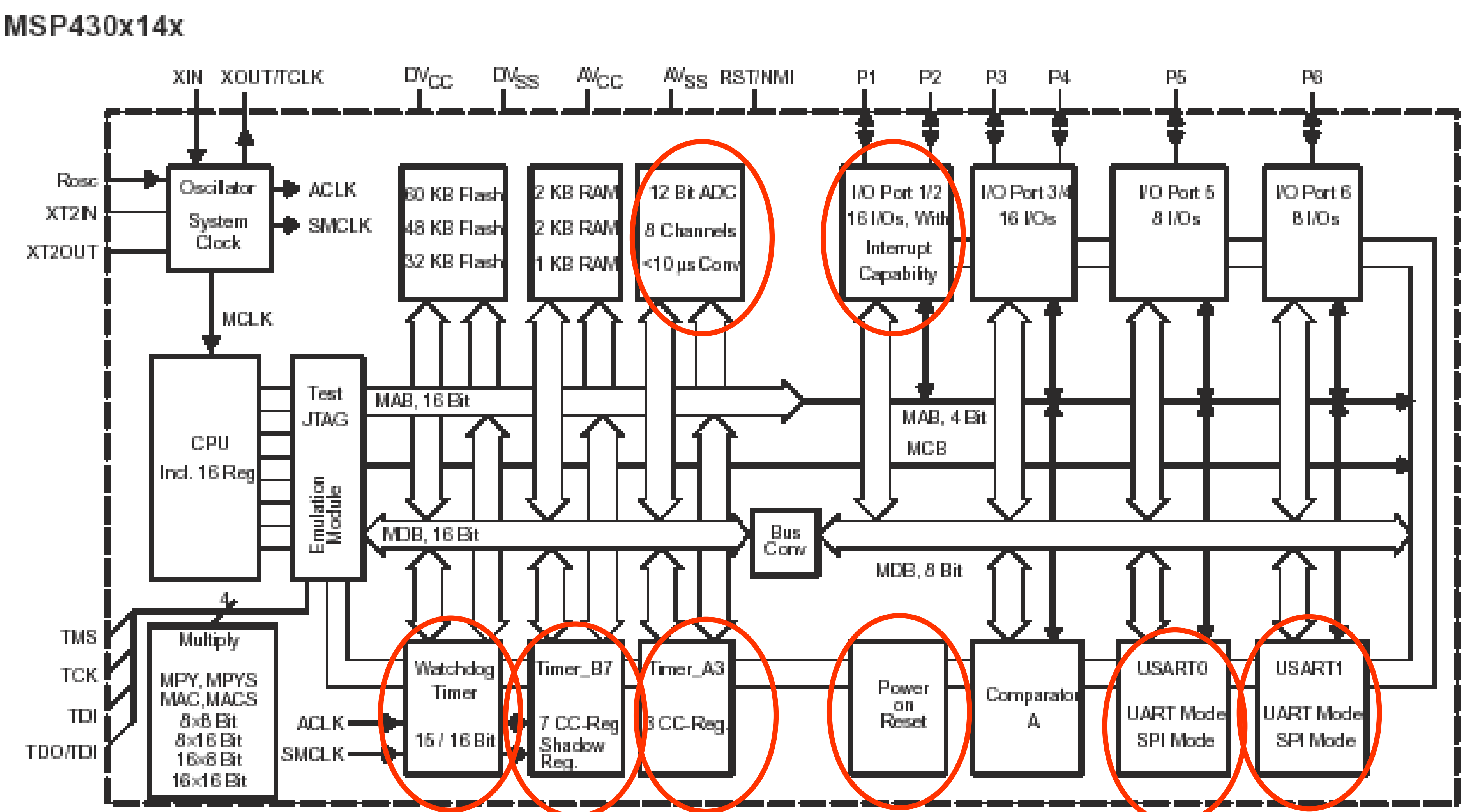
- Transparent Interrupt
  - Take actions and run code, but
  - When the dust settles, computer should be in exactly the same state as before the interrupt.
- Priority
  - When there are multiple I/O devices, potential for interrupts to occur during ISRs
  - Assign priority to interrupts and handle time-critical tasks first.

MSP430 Interrupt Mechanisms

- Peripheral devices
  - USART, Timers, AD/DA, GPIO
- Pins
  - NMI, Reset, P.0, P.1

- A daisy chain of requestors
- On MSP430:
  - Closer to processor, the higher priority
- Power-up : highest priority
- See priority table (datasheet).

MSP430 Interrupt Mechanisms



MSP430 Interrupt Processing

- Table definition:

```
ORG     0FFE0h
DW      Invalid_Interrupt

ORG     0FFE2h
DW      Invalid_Interrupt

ORG     0FFE4h
DW      Port1_Interrupt

ORG     0FFE6h
DW      Port2_Interrupt

ORG     0FFE8h
DW      Invalid_Interrupt
```

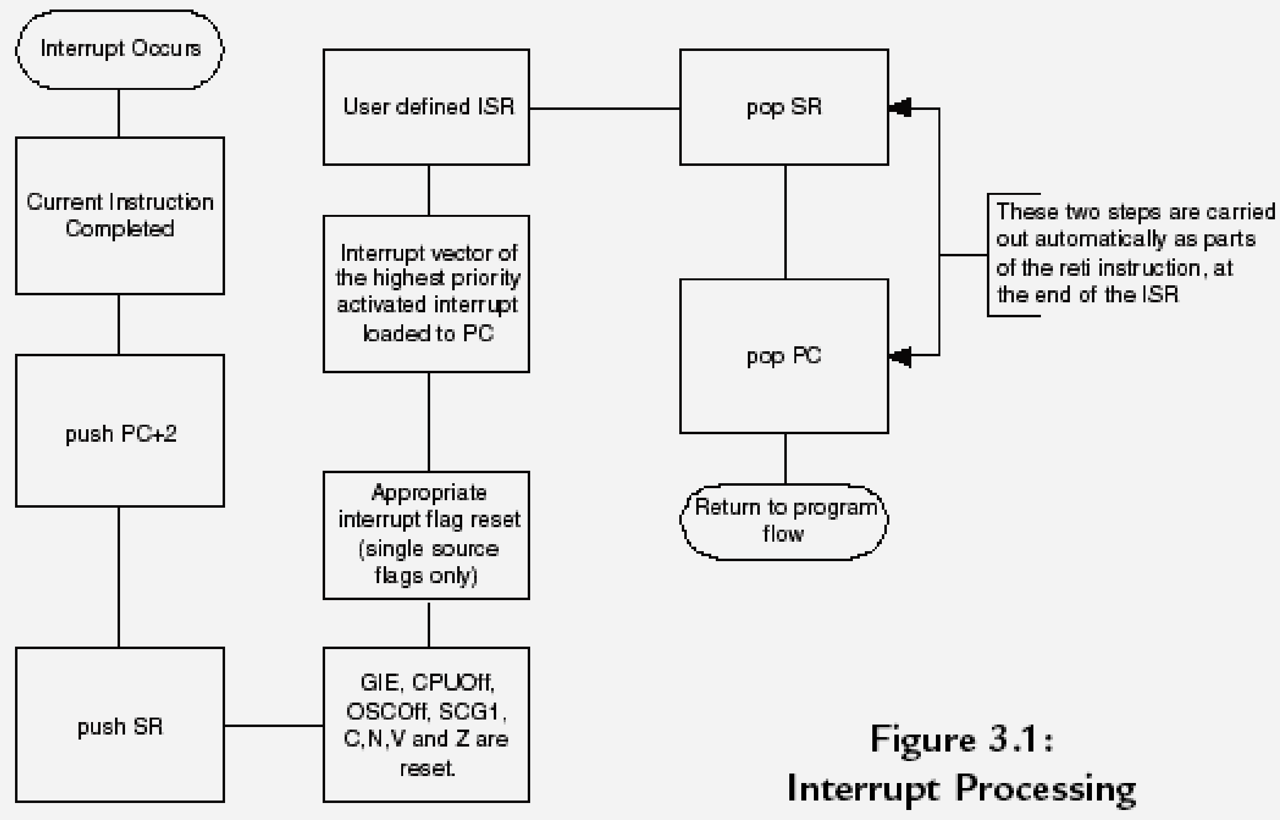
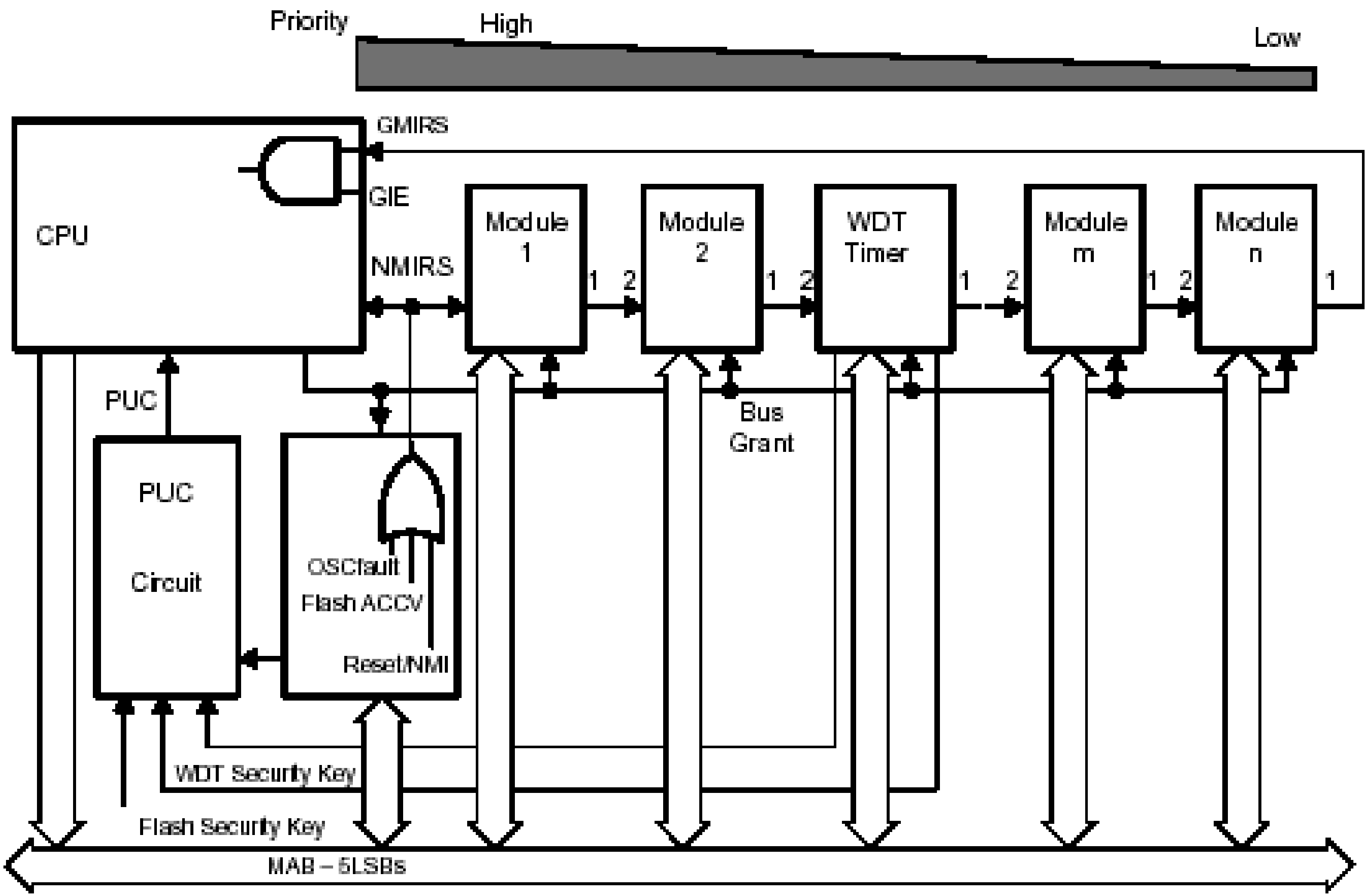


Figure 3.1: Interrupt Processing

MSP430 Interrupt Priorities



Interrupt Advice

- Always use RETI
- Ensure unused interrupts disabled
- Make ISRs fast (stack problems with nesting)
  - Only change flags, return to main
- Consider polling as alternative
  - stack overflow
  - race condition
  - boundary conditions
- ISRs in C
  - Example: few routines
    (IAR compiler convention)

```
interrupt [0x02] void dummy_Port2_Interrupt(void)
 {
 while (1)
   {}
 }
interrupt [0x04] void dummy_USART1tx_Interrupt(void)
 {
 while (1)
   {}
 }
```

Some Programming Tips

- More comprehensive next week, but to get you going
- Modular design
  - Use header files and comment them!!
  - Should be a driver for each peripheral
- Commenting is critical
  - Top of each function – explain inputs, outputs, purpose.
- Use the pre-defined labels for registers etc.

Embedded C tips

- C preprocessor
  - #define, #ifndef, #if, #ifdef, #else …etc
  - Used to prevent multiple includes
  - #ifndef _TIMER_H_
  - #define _TIMER_H_
  - #endif

C preprocessor

- Complex macros
  - Passing parameters
  - Padding of parameters
- Widely used in Firmware drivers

```
#define SET_VAL(x) LCD_Settings.P##x
#define SET(x, val) SET_VAL(x) = val
#define DEF_SET(x) SET(x, DS_P##x)

#define MIN(n,m) (((n) < (m)) ? (n) : (m))

#define MAX(n,m) (((n) < (m)) ? (m) : (n))

#define ABS(n) ((n < 0) ? -(n) : (n))
```

Global variables

- Distinguish global variables from local by choosing appropriate naming convention
  - Example: RX_Buffer_Gbl
  - Stick to your convention throughout the program

- Use global variables as Software flags
  - Example: PACKET_RECEIVED – use capitals

- Have them all in ONE place

Report Guidelines

- Section 1: Functional Specifications
- Section 2: Implementation
- Section 3: Performance Analysis

Functional Specifications

- **Function**: void posadd(int *a, int *b, int *c)
- **Purpose**: Adds two positive integers and writes the sum to a specified memory location.
- **Inputs**: a,b – pointers to memory locations storing two positive 16-bit integers in 2's-complement binary.
- **Output**: *c = *a + *b. Write result of sum to memory location pointed to by c. Result is 16-bit 2's complement binary.
- **Special Cases/Error Conditions**:
  - Negative input (*a<0 OR *b < 0) : set *c = -1.
  - Overflow (*a + *b > 2^15-1) : set *c = -2.

Functional Specifications

- Write for the user.
- No implementation details.
- Be precise!
- User should be able to look at functional specs and know exactly how to include your function.

Implementation

- Include a concise explanation of your solution.
- What is the flow of the program?
- What hardware is used? How is it used?
- What flags are set?
- How is configuration performed? How are interrupts handled?
- Answer any other questions you consider pertinent.
- If you have made decisions, include a brief justification/explanation. For example, why did you use a 10 ohm resistor instead of a 10 Kohm?

Implementation (2)

- If your deliverable involves a user interface, include a subsection discussing the user interface and clearly explain the reasoning behind each decision.
- e.g., why did you include a menu rather than a command line?
- Properly prepared flow diagrams are highly encouraged
- Carefully prepared diagrams are expected when new hardware is connected (Lab 4).

Performance Analysis

- Include a concise description of the tests you performed to verify correct behaviour and discuss the performance.
- Tables are often the best way to display your results (input, condition tested, output of program).
- For the labs, there will be other performance analysis that is appropriate (for example, in Lab 1 you should explore the number of cycles that your routines consume).

Performance Analysis (2)

- In later labs, where your deliverable involves a user interface, include a user survey.
- Poll at least 8 people, at least 4 from the lab and at least 2 from outside.
- Your poll should consist of at least 5 questions.
- Include the survey as an appendix to your report.
- Ask reasonable questions and provide a user response of 1-10.
- Include in your report a table specifying min/max and mean scores.
- Also include the most pertinent comments, e.g., “the menu system was very easy to learn, but became tiresome because it was slow when I had some expertise”.

Performance Analysis (3)

- Example (incomplete, but you should get the idea):
- Table 1: Performance verification of function library (correctness)

Function: Condition	**Input**	**Output**	**Cycles**
posadd: normal	*a =21, *b =22	*c = 43	1000
posadd: negative	*a = -2, *b = 3	*c = -1	500
posadd: negative	*a = 3, *b = -2	*c = -1	500
posadd: overflow	*a = 2^15-1 *b = 2^15-1	*c = -2	1500

Performance Analysis (4)

Table 2: Speed testing of function library (10 random cases per row of table).

Function: Condition	**Min. cycles**	**Max. cycles**	**Mean cycles**	**Variance**
posadd: normal	500	2200	1000	1e5
posadd: negative	300	1100	500	2e5
posadd: overflow	700	1800	1300	1e5

References

- Reference all material you have used.
- Where you make use of document [1] include such notation in the text.
- References should be in the format specified in Section 4 of the report guidelines.
- If you do not know information, seek it out; if you cannot find it after a reasonable search, then ask me or the teaching assistants.
- The fact that a user guide has no date or author on the front cover is no excuse for you not to track the information down.

Appendix - Code

- Code : very well documented, small-font (10 or 11 point), 2 pages per sheet (landscape).
- If you have reused unchanged code from previous labs, do NOT include it in your report (just retain the header files).
- Keep the code well-organized, so that I can quickly locate your new work.