Computational hardware

- Digital logic (CSE370/351)
  - Gates and flip-flops: glue logic, simple FSMs, registers
  - Two-level PLDs: FSMs, muxes, decoders

- Programmable logic devices (CSE370/352, CSE467)
  - Field-programmable gate arrays: FSMs, basic data-paths
  - Mapping algorithms to hardware

- Microprocessors (CSE378/352)
  - General-purpose computer
  - Instructions can implement complex control structures
  - Supports computations/manipulations of data in memory

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
  - Many systems allow firmware updates
- 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

<table>
<thead>
<tr>
<th>CPU</th>
<th>Memory</th>
<th>Display (with dual-port video RAM)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1/O (serial line, keyboard, mouse)</td>
<td>Disk</td>
<td>Network Interface</td>
</tr>
</tbody>
</table>

all the parts around the processor are usually required

standard interfaces

Typical task-specific architecture

<table>
<thead>
<tr>
<th>General Purpose 1/O</th>
<th>Microcontroller (CPU+mem+)</th>
<th>A/D-D/A Conversion</th>
</tr>
</thead>
<tbody>
<tr>
<td>Special I/O Device Driver</td>
<td>ROM</td>
<td>RAM</td>
</tr>
<tr>
<td>ROM</td>
<td>RAM</td>
<td>Custom Logic</td>
</tr>
</tbody>
</table>

medium-speed interactions

low-speed interactions

high-speed interactions

standard interface

any of the parts around the microcontroller are optional
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 occurred
  - 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 MSP430: Introduction

MSP430: An Introduction

- The MSP430 family
- Technology Roadmap
- Typical Applications
- The MSP430 Documentation
- MSP430 Architecture
- MSP430 Devices
- MSP430 RISC core
The Family

- Broad family of TI’s 16-bit microcontrollers
  - from 1Kbytes ROM, 128 bytes RAM (approx. $1)
  - to 256Kbytes ROM, 16Kbytes RAM ($10)
- Many subfamilies
  - MSP430x1xx: Flash/ROM based MCUs offer 1.8V to 3.6V operation, up to 60kB, 8MIPS with Basic Clock.
  - **MSP430F2xx**: 16 MHz. integrated on-chip oscillator, internal pullup/pull-down resistors
  - MSP430x4xx: 120kB/Flash/ROM 8MIPS with FLL + SVS, integrated LCD controller
  - MSP430x5xx: 25 MIPS, 1.8 to 3.6V, Power Management Module for optimizing power consumption, 2x memory

Part numbering convention
MSP 430 Roadmap

MSP430 Typical Applications

Handheld Measurement
- Air Flow measurement
- Alcohol meter
- Barometer
- Data loggers
- Emission/Gas analyser
- Humidity measurement
- Temperature measurement
- Weight scales

Medical Instruments
- Blood pressure meter
- Blood sugar meter
- Breath measurement
- EKG system

Utility Metering
- Gas Meter
- Water Meter
- Heat Volume Counter
- Heat Cost Allocation
- Electricity Meter
- Meter reading system (RF)

Sports equipment
- Altimeter
- Bike computer
- Diving watches

Security
- Glass break sensors
- Door control
- Smoke/fire/gas detectors

Home environment
- Air conditioning
- Control unit
- Thermostat
- Boiler control
- Shutter control
- Irrigation system
- White goods (Washing machine,..)

Misc
- Smart card reader
- Taxi meter
- Smart Batteries
An MSP430-Based System

- LCD
- RS232 controller
- Analog I/O
- 2-axes joystick
- LEDs
- Switches
- Thermistor
- Keypad

Another MSP430-Based System

- CC430F6137 MCU
- <1GHz RF
  - 433, 868 & 915 MHz
- 2-Wire JTAG Access
- 96 segment LCD
- Buzzer
- eZ430 Programmer
- RF Access Point
- Chronos Disassembly Tool
- 3-Axis Accelerometer
- Pressure & Altitude Sensor
- Temperature Sensor
- Voltage & Battery Sensor
- CR2032 Battery
Chronos | Teardown

CC430 | Low-Power RF + Ultra-Low Power MCU

MSP430™ Microcontroller
- Industry's lowest power MCU
- 16-bit RISC architecture
- 27 MHz processor
- High-performance analog
- Sensor interface

CC1101 RF Transceiver SoC
- High sensitivity
- Low current consumption
- Excellent blocking performance
- Flexible data rate & modulation format

Intelligent Peripherals
- 100 nA comparator
- 8ch 12-bit ADC offering 200-kmps
- 96 segment LCD controller
- 128-bit AES security encryption/decryption coprocessor

64QFN Pin Package
- 9.1 mm x 9.1 mm area
MSP430 Documentation

- MSP430 home page (TI)
  - [www.ti.com/msp430](http://www.ti.com/msp430)

- User’s manual (MSP430x2xx Family)
  - [http://www.ti.com/litv/pdf/slau144e](http://www.ti.com/litv/pdf/slau144e)

- Datasheet

- Chronos:

MSP 430 Modular Architecture

- Architecture reduces power consuming, noise generating fetches to memory
- 16-bit bus handles wide-width data much more effectively
- Embedded emulation accessed in-application with JTAG
- van-Neumann common bus connects CPU to all memory and peripherals
CPU Introduction

- RISC architecture with 27 instructions and 7 addressing modes.
- Orthogonal architecture with every instruction usable with every addressing mode.
- Full register access including program counter, status registers, and stack pointer.
- Single-cycle register operations.
- Large 16-bit register file reduces fetches to memory.
- 16-bit address bus allows direct access and branching throughout entire memory range.
- 16-bit data bus allows direct manipulation of word-wide arguments.
- Constant generator provides six most used immediate values and reduces code size.
- Direct memory-to-memory transfers without intermediate register holding.
- Word and byte addressing and instruction formats.

MSP430 16-bit RISC

- Large 16-bit register file eliminates single accumulator bottleneck
- High-bandwidth 16-bit data and address bus with no paging
- RISC architecture with 27 instructions and 7 addressing modes
- Single-cycle register operations with full-access
- Direct memory-memory transfer designed for modern programming
- Compact silicon 30% smaller than an '8051 saves power and cost
CPU Registers

- R0 - PC Program Counter
  16-bit = no paging
- R1 - SP Stack Pointer
  Addressable = great "C" code
- R2 - SR Status Register
  Define LPMx
- R3/R2 - CG Constant Generator
  automatic generation of common used values reduces code size 30%

R4 through R15 are single-cycle, general purpose and identical in all respects - used for math, storage, and addressing modes.

Registers: PC (R0)

- Each instruction uses an even number of bytes (2, 4, or 6)
- PC is word aligned (the LSB is 0)

MOV #LABEL,PC ; Branch to address LABEL
MOV LABEL,PC ; Branch to address contained in LABEL
MOV @R14,PC ; Branch indirect, indirect R14
 Registers: SP (R1)

- Stack pointer for return addresses of subroutines and interrupts
- SP is word aligned (the LSB is 0)
- Pre-decrement/post-increment scheme

```plaintext
MOV 2(SP), R6 ; Item I2 -> R6
MOV R7, 0(SP) ; Overwrite TOS with R7
PUSH #0123h ; Put 0123h onto TOS
POP R8 ; R8 = 0123h
```

 Registers: SR (R2)

- C: SR(0)
- Z: SR(1)
- N: SR(2)
- GIE (Global interrupt enable): SR(3)
- CPUOff: SR(4)
- OSCOff: SR(5)
- SCG1, SCG0: SR(7), SR(6)
- V: SR(8)
### Status bits

<table>
<thead>
<tr>
<th>Bit</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>V</td>
<td>Overflow bit. This bit is set when the result of an arithmetic operation overflows the signed-integer range.</td>
</tr>
<tr>
<td>ADD (B), ADDC (B)</td>
<td>Set when: Positive + Positive = Negative, Negative + Negative = Positive, otherwise reset</td>
</tr>
<tr>
<td>SUB (B), SUBC (B), CMP (B)</td>
<td>Set when: Positive + Negative = Negative, Negative + Positive = Positive, otherwise reset</td>
</tr>
<tr>
<td>SCG1</td>
<td>System clock generator 1. This bit, when set, turns off the SMCLK.</td>
</tr>
<tr>
<td>SCG0</td>
<td>System clock generator 0. This bit, when set, turns off the DCO dc generator, if DCOCLK is not used for MCLK or SMCLK.</td>
</tr>
<tr>
<td>OSCOFF</td>
<td>Oscillator Off. This bit, when set, turns off the LFXT1 crystal oscillator, when LFXT1CLK is not used for MCLK or SMCLK.</td>
</tr>
<tr>
<td>CPUOFF</td>
<td>CPU off. This bit, when set, turns off the CPU.</td>
</tr>
<tr>
<td>GIE</td>
<td>General interrupt enable. This bit, when set, enables maskable interrupts. When reset, all maskable interrupts are disabled.</td>
</tr>
<tr>
<td>N</td>
<td>Negative bit. This bit is set when the result of a byte or word operation is negative and cleared when the result is not negative. Word operation: N is set to the value of bit 15 of the result. Byte operation: N is set to the value of bit 7 of the result.</td>
</tr>
<tr>
<td>Z</td>
<td>Zero bit. This bit is set when the result of a byte or word operation is 0 and cleared when the result is not 0.</td>
</tr>
<tr>
<td>C</td>
<td>Carry bit. This bit is set when the result of a byte or word operation produces a carry and cleared when no carry occurred.</td>
</tr>
</tbody>
</table>

### Constant Generators

- **As – source register addressing mode** in the instruction word

<table>
<thead>
<tr>
<th>Register</th>
<th>As</th>
<th>Constant</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>R2</td>
<td>00</td>
<td>—— ——</td>
<td>Register mode</td>
</tr>
<tr>
<td>R2</td>
<td>01</td>
<td>(0)</td>
<td>Absolute address mode</td>
</tr>
<tr>
<td>R2</td>
<td>10</td>
<td>00004h</td>
<td>+4, bit processing</td>
</tr>
<tr>
<td>R2</td>
<td>11</td>
<td>00008h</td>
<td>+8, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>00</td>
<td>00000h</td>
<td>0, word processing</td>
</tr>
<tr>
<td>R3</td>
<td>01</td>
<td>00001h</td>
<td>+1</td>
</tr>
<tr>
<td>R3</td>
<td>10</td>
<td>00002h</td>
<td>+2, bit processing</td>
</tr>
<tr>
<td>R3</td>
<td>11</td>
<td>0000Fh</td>
<td>—1, word processing</td>
</tr>
</tbody>
</table>
CISC / RISC Instruction Set

- Three instruction formats
  - Source, destination
  - Destination
  - Jumping

- Fifty-one instructions available in assembler
  - 27 basic instructions ⇒ RISC
  - 24 emulated instructions ⇒ CISC

- Seven addressing modes for source, four for destination
  - Register Mode
  - Indexed Mode
  - Symbolic Mode
  - Absolute Mode
  - Indirect Mode
  - Indirect-autoincrement Mode
  - Immediate Mode

- Bit, byte and word processing

---

27 Core RISC Instructions

<table>
<thead>
<tr>
<th>Format I</th>
<th>Format II</th>
<th>Format III</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source, Destination</td>
<td>Single Operand</td>
<td>+/- 9bit Offset</td>
</tr>
<tr>
<td>add(.b)</td>
<td>call</td>
<td>jce</td>
</tr>
<tr>
<td>addc(.b)</td>
<td>nemp</td>
<td>jnc</td>
</tr>
<tr>
<td>and(.b)</td>
<td>ncr</td>
<td>jnc</td>
</tr>
<tr>
<td>bic(.b)</td>
<td>push(.b)</td>
<td>jeq</td>
</tr>
<tr>
<td>bis(.b)</td>
<td>ret</td>
<td>jne</td>
</tr>
<tr>
<td>bit(.b)</td>
<td>rra(.b)</td>
<td>jge</td>
</tr>
<tr>
<td>cmp(.b)</td>
<td>rrc(.b)</td>
<td>jl</td>
</tr>
<tr>
<td>dadd(.b)</td>
<td></td>
<td>jn</td>
</tr>
<tr>
<td>mov(.b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sub(.b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>subc(.b)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>xor(.b)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Emulated Instructions

- Simply easier to understand with no code size or speed penalty
- Replaced by assembler with core instructions using CG, PC and SP

```
clrc
bic.w #01h,SR ; Clear carry (emulated)
    ; Core instruction

dec.w R4
sub.w #01,R4 ; Decrement (emulated)
    ; Core instruction

ret
mov.w @SP+,PC ; Return (emulated)
    ; Core instruction
```

51 Total Instructions

<table>
<thead>
<tr>
<th>Format I Source, Destination</th>
<th>Format II Single Operand</th>
<th>Format III +/- 9bit Offset</th>
<th>Support</th>
</tr>
</thead>
<tbody>
<tr>
<td>add.h</td>
<td>br</td>
<td>jmp</td>
<td>clrc</td>
</tr>
<tr>
<td>addc.h</td>
<td>call</td>
<td>jc</td>
<td>setc</td>
</tr>
<tr>
<td>and.h</td>
<td>swpb</td>
<td>jnc</td>
<td>clrz</td>
</tr>
<tr>
<td>bic.h</td>
<td>ext</td>
<td>jeq</td>
<td>setz</td>
</tr>
<tr>
<td>bis.h</td>
<td>push.h</td>
<td>jpe</td>
<td>clrn</td>
</tr>
<tr>
<td>bit.h</td>
<td>pop.h</td>
<td>jge</td>
<td>satn</td>
</tr>
<tr>
<td>cmp.h</td>
<td>rra.h</td>
<td>jl</td>
<td>dint</td>
</tr>
<tr>
<td>dec.h</td>
<td>rrc.h</td>
<td>jn</td>
<td>sint</td>
</tr>
<tr>
<td>mov.h</td>
<td>inv.h</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>sub.h</td>
<td>inc.h</td>
<td>ret</td>
<td></td>
</tr>
<tr>
<td>subc.h</td>
<td>incdi.h</td>
<td>retl</td>
<td></td>
</tr>
<tr>
<td>xor.h</td>
<td>dec.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>decd.h</td>
<td>addc.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sbc.h</td>
<td>clrc.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>daddc.h</td>
<td>dra.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rla.h</td>
<td>rlc.h</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tst.h</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Double operand instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV (.B)</td>
<td>src, dst</td>
<td>src -&gt; dst</td>
<td>V N Z C</td>
</tr>
<tr>
<td>ADD (.B)</td>
<td>src, dst</td>
<td>src + dst -&gt; dst</td>
<td>* * * *</td>
</tr>
<tr>
<td>ADDC (.B)</td>
<td>src, dst</td>
<td>src + dst + C -&gt; dst</td>
<td>* * * *</td>
</tr>
<tr>
<td>STB (.B)</td>
<td>src, dst</td>
<td>dst + not.src + 1 -&gt; dst</td>
<td>* * * *</td>
</tr>
<tr>
<td>STBC (.B)</td>
<td>src, dst</td>
<td>dst + not.src + C -&gt; dst</td>
<td>* * * *</td>
</tr>
<tr>
<td>CMP (.B)</td>
<td>src, dst</td>
<td>dst - src</td>
<td>* * * *</td>
</tr>
<tr>
<td>DADD (.B)</td>
<td>src, dst</td>
<td>src + dst + C -&gt; dst (decimal)</td>
<td>* * * *</td>
</tr>
<tr>
<td>BIT (.B)</td>
<td>src, dst</td>
<td>src .and. dst</td>
<td>0 * * *</td>
</tr>
<tr>
<td>BIC (.B)</td>
<td>src, dst</td>
<td>.not.src .and dst</td>
<td>0 * * *</td>
</tr>
<tr>
<td>BIS (.B)</td>
<td>src, dst</td>
<td>src .or. dst</td>
<td>0 * * *</td>
</tr>
<tr>
<td>XOR (.B)</td>
<td>src, dst</td>
<td>src .xor. dst</td>
<td>0 * * *</td>
</tr>
<tr>
<td>AND (.B)</td>
<td>src, dst</td>
<td>src .and. dst</td>
<td>0 * * *</td>
</tr>
</tbody>
</table>

CSE 466 Microcontrollers 49

Single Operand Instruction

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
<th>Status Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC (.B)</td>
<td>dst</td>
<td>C -&gt; MSB -&gt; ... -&gt; LSB -&gt; C</td>
<td>* * * *</td>
</tr>
<tr>
<td>RRA (.B)</td>
<td>dst</td>
<td>MSB -&gt; MSB -&gt; ... -&gt; LSB -&gt; C</td>
<td>0 * * *</td>
</tr>
<tr>
<td>POSR (.B)</td>
<td>src, dst</td>
<td>SP -&gt; SP, src -&gt; @SP</td>
<td>0 * * *</td>
</tr>
<tr>
<td>SWPB</td>
<td>dst</td>
<td>Swap bytes</td>
<td>0 * * *</td>
</tr>
<tr>
<td>CALL</td>
<td>dst</td>
<td>SP -&gt; SP, PC+2 -&gt; @SP</td>
<td>0 * * *</td>
</tr>
<tr>
<td>RETI</td>
<td>TOS -&gt; SR, SP + 2 -&gt; SP</td>
<td>0 * * *</td>
<td></td>
</tr>
<tr>
<td>SXT</td>
<td>dst</td>
<td>Bit 7 -&gt; Bit 8 ... Bit 15</td>
<td>0 * * *</td>
</tr>
</tbody>
</table>
Jump Instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>S-Reg, D-Reg</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>JEQ/JZ</td>
<td>Label</td>
<td>Jump to label if zero bit is set</td>
</tr>
<tr>
<td>JNE/JNZ</td>
<td>Label</td>
<td>Jump to label if zero bit is reset</td>
</tr>
<tr>
<td>JC</td>
<td>Label</td>
<td>Jump to label if carry bit is set</td>
</tr>
<tr>
<td>JNC</td>
<td>Label</td>
<td>Jump to label if carry bit is reset</td>
</tr>
<tr>
<td>JN</td>
<td>Label</td>
<td>Jump to label if negative bit is set</td>
</tr>
<tr>
<td>JGE</td>
<td>Label</td>
<td>Jump to label if (N XOR V) = 0</td>
</tr>
<tr>
<td>JL</td>
<td>Label</td>
<td>Jump to label if (N XOR V) = 1</td>
</tr>
<tr>
<td>JMP</td>
<td>Label</td>
<td>Jump to label unconditionally</td>
</tr>
</tbody>
</table>

3 Instruction Formats

; Format I Source and Destination

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad/BW</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>5405</td>
<td>add.w R4,R5</td>
<td></td>
<td></td>
<td>R4+R5=R5 xxxx</td>
</tr>
<tr>
<td>5445</td>
<td>add.b R4,R5</td>
<td></td>
<td></td>
<td>R4+R5=R5 0000</td>
</tr>
</tbody>
</table>

; Format II Destination Only

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>B/W</th>
<th>Ad</th>
<th>DS-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>6404</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6444</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

; Format III There are 8 (Un)conditional Jumps

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Condition</th>
<th>10-bit PC Offset</th>
</tr>
</thead>
<tbody>
<tr>
<td>3c28</td>
<td>jmp</td>
<td>Loop_1</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Goto Loop_1</td>
</tr>
</tbody>
</table>
Addressing Modes

<table>
<thead>
<tr>
<th>As/Ad</th>
<th>Addressing Mode</th>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00/0</td>
<td>Register mode</td>
<td>Rn</td>
<td>Register contents are operand</td>
</tr>
<tr>
<td>01/1</td>
<td>Indexed mode</td>
<td>X(Rn)</td>
<td>(Rn + X) points to the operand. X is stored in the next word.</td>
</tr>
<tr>
<td>01/1</td>
<td>Symbolic mode</td>
<td>ADDR</td>
<td>(PC + X) points to the operand. X is stored in the next word. Indexed mode X(PC) is used.</td>
</tr>
<tr>
<td>01/1</td>
<td>Absolute mode</td>
<td>&amp;ADDR</td>
<td>The word following the instruction contains the absolute address. X is stored in the next word. Indexed mode X(SR) is used.</td>
</tr>
<tr>
<td>10/-</td>
<td>Indirect register mode</td>
<td>@Rn</td>
<td>Rn is used as a pointer to the operand.</td>
</tr>
<tr>
<td>11/-</td>
<td>Indirect autoincrement</td>
<td>@Rn+</td>
<td>Rn is used as a pointer to the operand. Rn is incremented afterwards by 1 for .B instructions and by 2 for .W instructions.</td>
</tr>
<tr>
<td>11/-</td>
<td>Immediate mode</td>
<td>#N</td>
<td>The word following the instruction contains the immediate constant N. Indirect autoincrement mode @PC+ is used.</td>
</tr>
</tbody>
</table>

Register Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0100</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0101</td>
</tr>
</tbody>
</table>

4405  mov.l R4, R5 ;
4445  mov.b R4, R5 ;

Valid for Source and destination As=00, Ad=0
The operand is contained in one of the CPU registers R0 to R15. This is the fastest addressing mode and needs the least memory.
Register-Indexed Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>BW</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0100</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0101</td>
</tr>
</tbody>
</table>

449501000200 mov.w 100h(R4),200h(R5)
44150100 mov.w 100h(R4),R5

Valid for Source and destination As=01, Ad=1
The address of the operand is the sum of the index and the contents of the register.

Symbolic Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>BW</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0000</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0000</td>
</tr>
</tbody>
</table>

409fffa80006 mov.w EDE,TONI
4015ffac mov.w EDE,R5

Source and destination As=01, Ad=1
The content of the addresses EDE / TONI are used for the operation. The source or destination address is computed as a difference from the PC and uses the PC in indexed addressing mode. Any address in the 64k memory space is addressable.
### Absolute Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0010</td>
<td>1</td>
<td>0</td>
<td>01</td>
<td>0010</td>
</tr>
</tbody>
</table>

429201720174 \texttt{mov.w} &CCR0,&CCR1 ;

42150172 \texttt{mov.w} &CCR0,R5 ;

**Source and destination As=01, Ad=1**
The contents of the fixed addresses are used for the operation. The SR is used in the indexed mode to create an absolute 0. Use for hardware peripherals located at an absolute address that can never be relocated.

---

### Register Indirect Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0100</td>
<td>0</td>
<td>0</td>
<td>10</td>
<td>0101</td>
</tr>
</tbody>
</table>

4425 \texttt{mov.w} @R4,R5 ;

4465 \texttt{mov.b} @R4,R5 ;

**Source only As=10, Ad=n/a**
The registers are used as a pointer to the operand. The indexed mode with zero index may be used for "indirect register addressing" of the destination operand.

44a50000 \texttt{mov.w} @R4,0(R5) ;
# Register Indirect

## Autoincrement Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0100</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0101</td>
</tr>
</tbody>
</table>

4435      mov.w   @R4+,R5    ;
4475      mov.b   @R4+,R5    ;

*Source only As=11, Ad=n/a*

The registers are used as a pointer to the operand. The registers are incremented afterwards - by 1 in byte mode, by 2 in word mode.

---

# Immediate Addressing Mode

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>Source-Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0000</td>
<td>0</td>
<td>0</td>
<td>11</td>
<td>0101</td>
</tr>
</tbody>
</table>

49351234  mov.w   #1234h,R5  ; Any 16-bit value

*Source only As=11, Ad=n/a*

Any immediate 8 or 16 bit constant can be used with the instruction. The **PC** is used in autoincrement mode to emulate this addressing mode.
**Code Reduction Effect of Constant Generator**

<table>
<thead>
<tr>
<th>OpCode</th>
<th>Source-Register</th>
<th>Ad</th>
<th>B/W</th>
<th>As</th>
<th>Destination-Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>0011</td>
<td>0</td>
<td>0</td>
<td>01</td>
<td>0100</td>
</tr>
</tbody>
</table>

```
4314  mov.w  #0001h,R4
```

R3/R2 - CG Constant Generator

automatic generation of commonly used values reduces code size 30%

---

**Machine Cycles for Format I Instructions**

<table>
<thead>
<tr>
<th>Address Mode</th>
<th># of Cycles</th>
<th>Length of Instruction [words]</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>00, Rr</td>
<td>0, Rn</td>
<td>1</td>
<td>MOV R5,R8 BR R9</td>
</tr>
<tr>
<td>00, Rr</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>ADD R5,2(R6) XOR R8,EDE MOV R5,EDE</td>
</tr>
<tr>
<td>01, x(Rn)</td>
<td>0, Rs</td>
<td>2</td>
<td>MOV R5,R7 AND EDE,R4</td>
</tr>
<tr>
<td>01, x(Rn)</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>ADD 4(R4),(6(R3)) CMP EDE,TIN</td>
</tr>
<tr>
<td>01, x(Rn)</td>
<td>1, TIN()</td>
<td>6</td>
<td>MOV R5,EDE</td>
</tr>
<tr>
<td>10, @Rr</td>
<td>0, Rm</td>
<td>2</td>
<td>AND @R8,E8</td>
</tr>
<tr>
<td>10, @Rr</td>
<td>1, EDE</td>
<td>5</td>
<td>XOR @R5,E8 MOV @R5,EDE XOR @R5,E8</td>
</tr>
<tr>
<td>11, @Rr</td>
<td>0, Rm</td>
<td>2</td>
<td>ADD @R5,R6 BR @R5+ MOV #R5,R9 BR #2Aeh</td>
</tr>
<tr>
<td>11, x(Rn)</td>
<td>0, Rm</td>
<td>3</td>
<td>MOV @R5+2(R4)</td>
</tr>
<tr>
<td>11, x(Rn)</td>
<td>1, EDE</td>
<td>5</td>
<td>ADD #31,EDE</td>
</tr>
<tr>
<td>11, @Rr</td>
<td>1, EDE</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
<tr>
<td>11, x(Rn)</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
<tr>
<td>11, @Rr</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
<tr>
<td>11, @Rr</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
<tr>
<td>11, x(Rn)</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
<tr>
<td>11, @Rr</td>
<td>1, x(Rn)</td>
<td>2</td>
<td>MOV @R5+EDE</td>
</tr>
</tbody>
</table>
Machine Cycles for Format II/III Instructions

<table>
<thead>
<tr>
<th>Address Mode</th>
<th># of Cycles</th>
<th>Length of Instruction [words]</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>00, Rn</td>
<td>1</td>
<td>3/4</td>
<td>SWPB RS</td>
</tr>
<tr>
<td>01, x(Rn)</td>
<td>4</td>
<td>5/5</td>
<td>CALL Table(R7) PUSH EDE</td>
</tr>
<tr>
<td>01, EDE</td>
<td>5</td>
<td>4/4</td>
<td>RRC @R9</td>
</tr>
<tr>
<td>11, @Rn+</td>
<td>3</td>
<td>4/5</td>
<td>SWPB @(R1)+ CALL 2(R7)</td>
</tr>
<tr>
<td>11, 4N</td>
<td>3</td>
<td>4/5</td>
<td></td>
</tr>
</tbody>
</table>

Machine Cycles for Format III Instructions

All Jxx - instructions need the same # of cycles independent of executing a Jump

- Clock Cycles: 2
- Length of Instruction: 1 word

MSP430 Memory Model

- Unified 64kB continuous memory map
- Same instructions for data and peripherals
- Program and data in Flash or RAM with no restrictions
- Easy to understand with no paging
- Designed for modern programming techniques such as pointers and fast look-up tables
Memory Organization

MSP 430 Architecture: A Closer Look
MSPx430x14x Architecture

64 TQFP (The Thin Quad Flat Pack package)

Basic Clock System

Basic Clock Module provides the clocks for the MSP430 devices
**Watchdog Timer**

WDT module performs a controlled system restart after a software problem occurs

- Can serve as an interval timer (generates interrupts)
- WDT Control register is password protected
- Note: Powers-up active

**Timer_A**

Timer_A is a 16-bit timer/counter with three capture/compare registers

- Capture external signals
- Compare PWM mode
- SCCI latch for asynchronous communication
**Comparator_A**

Comparator_A is an analog voltage comparator

- Supports precision slope analog-to-digital conversions
- Supply voltage supervision, and
- Monitoring of external analog signals.

---

**Digital I/O**

Independently programmable individual I/Os

- Up to 6 ports (P1 – P6)
- Each has 8 I/O pins
- Each pin can be configured as input or output
- P1 and P2 pins can be configured to assert an interrupt request
**ADC12**

High-performance 12-bit analog-to-digital converter

- More than 200 Ksamples/sec
- Programmable sample & hold
- 8 external input channels
- Internal storage

**USART Serial Port**

The universal synchronous/asyncronous receive/transmit (USART) peripheral interface supports two serial modes with one hardware module

- UART or SPI (Synchronous Peripheral Interface) modes
- Double-buffered
- Baud-rate generator