Introduction to Microprocessors

Shahzeb Asif

June 17, 2017

fundamentals

microprocessors

- the brains of a computer on a single integrated circuit
- basically a sequential state machine

application of microprocessors

- everywhere
  - general purpose
  - special purpose
  - embedded
    * for a dedicated task

history

- intel 4004 in 1971
- amd came later
- motorola made coldfire
- moore's law
  - postulated in 75
  - predicts doubling of transistors every couple of years

advantages of microprocessors

- revolutionized digital systems
- programmable for general purpose
- compact
- performance was great
- cost effective
- reliable

disadvantages of microprocessors

- steep learning curve
- unforgiving in terms of mistakes
- development equipment, time, and money needed to work

terminology

- CPU: central processing unit
- muP: microprocessor
- muPU: microprocessor unit
- microcomputer: system built on a microprocessor
- MCU: microcontroller unit with extra circuitry
- DSP: digital signal processing. better for numerical intensive operations

microprocessor family

- intel
  - (71), 8008 (72), ... , core i7 (2008)
- motorola
  - (74), 6809 (79), ... , coldfire (94)
- amd
  - athelon (01), ... , phenom II (09)
- generally, these manufacturers have upwards/forward compatibility meaning that the future processors will have all the functionality of the previous ones
- we use the MCF 5270 MCU
  - it has features we don't use

single bus architecture

- von meumann
- there are three buses
  - address bus
  - data bus
  - control bus
- there are three components
  - cpu
  - main memory
  - i/o devices
- only the cpu writes to the address bus

system clocks

- clocks are to synchronize the different components
- the higher the clock speed, the more instructions can be executed
- overlocking is making the clock frequency higher than the recommended frequency

system buses

- bus is a group of wires
  - e.g. 4 bit data bus width of the data bus is 4
- the three primary buses
  - address bus
    - used by the cpu to tell the system what memory location is being used
  - data bus
    - used to communicate information
    - read data from device to cpu
    - write data from cpu to device
    - data bus width affects the amount of information that can be transferred at a time
  - control bus
    - coordinate the address and data bus
    - signals controlled by cpu
    - has signals such as: R/W, WE, OE

bus operations

- types of sequences
  - read cycle
  - write cycle
  - read modify write cycles
  - interrupt acknowledgement cycle
- an instruction can take more than one bus cycle

von neumann features

- advantages
  - simple
  - program is data
  - simple assembler
  - simple compiler
- disadvantages
  - limited efficiency
  - slow input/output

two bus architecture

- harvard architecture
- takes care of the bottle neck
- uses two buses to speed up system
- one bus is attached to the i/o memory and data memory and the other bus is attached to the program memory
- von neumann can emulate harvard with caches

elements of a cpu

programmer's model

- program counter (PC)
  - keeps track of the address of the next instruction
- data register (Dx)
  - stores operands and results
- address register (Ax)
  - for stroing addresses of operands and results
- status register (SR)
  - keeps summary of result of an operation like carry and stuf
- control register (CR)
  - often combined with SR
  - controls things like sleep, interrupts

invisible side

- instruction register (IR)
  - the instructions are decoded to find purpose
- arithmatic and logic unit (ALU)
  - manipulates data and addresses according to instructions
- bus interface logic
  - control
  - address
  - data
- others
  - exception and interrupt handling
  - memory management
  - floating point unit
  - branch prediction logic
- register: a piece of internal RAM that is implicitly referred to rather than bus interface logic

execution of an instruction

- examples: MOVE, ADD, SUB, SWAP
- cycle
  - fetch instruction
    - the PC value is placed on the address bus
    - read the instruction into the IR
  - decode the instruction
    - what needs to be done
  - fetch more info (optional)
    - get more information
  - execute
    - perform operation
    - all cpu registers are updated
    - increment PC
    - execution may take more than one cpu cycle

memory

- memory holds info in binary
- location of info is specified by address
- number of bits for each piece of info is device specific
- RAM: random access memory
  - info can be written or read
  - types
    - SRAM: static RAM (fast, expensive, static)
    - DRAM: dynamic RAM (slow, cheap, refreshes)
- ROM: read only memory
  - info can only be read
    - e.g. EEPROM, EPROM, Flash, Mask ROM
  - memory map
    - an 8 x 4 piece of memory has 8 addresses with 4 bits assigned to each address
    - number of address lines
      - each address needs a unique id
      - similar to last semester $2; n > num_{addresses}$
      - e.g. 8 addresses require 3 address lines
    - number of data lines
      - depends on how many bits each address is assigned to

input and output devices

- these actually interface with the world
- input examples
  - keyboard
  - mouse
  - A/D converter
- output examples
  - monitor
  - printer
  - D/A converter
- both
  - ethernet
  - usb
- devices like these are often connected via its address and data buses
- they are basically mapped to memory

register transfer language (RTL)

- shorthand for assembly concepts
- `[]` denotes the contents
  - e.g. `[PC]` are the contents of the program counter
  - e.g. `[M(0x7234)]` are the contents of memory location $0x7234_{16}$
- () indicates a memory location
  - e.g. M(100) is the memory location 100_{10}
- example: `[PC] <- 4` means places value 4 in the PC register

numerical data representation

- signed and unsigned numbers
- codes
  - ascii
  - bcd

positional number system

- basically:
  - $12 = 1 * 10; 1 + 2 * 10; 0$
- it continue on the other side of the radix point (decimal point)
- review from ece 210
  - divide by new base and read remainders up
  - multiply by new base and read whole numbers down

precision

- number of digits required for a certain precision depends on base
- formula: $\frac{1}{R;k} < precision_{10}$
- example: $\frac{1}{2;k} < 0.1_{10}$ $\frac{1}{2;4} < 0.1$
- in essence, it's the largest k needed to make the inequality true

review 210

- converting between binary, octal, decimal, and hexadecimal
- bcd

a note on values

- values are real but the way we represent them is an abstraction

data width

- size of memory dictates what range of numbers can be stores there
- signed $-2;n-1 \leq x \leq 2;n-1-1$
- unsigned $0 \leq x \leq 2;n-1$
- data sizes
  - nibble is 4 bits
  - byte is 8 bits
  - word is 16 bits
  - long is 32 bits
- 's complement
- conversion from signed to unsigned

binary addition

- add with the same rules as in 210
- c implies unsigned overflow
- v implies signed overflow
- don't forget them

data codes

- data codes make it convenient to convert data easily
- ASCII is for strings
  - e.g. subtract 0x20 from a lower case letter to make it upper case
- BCD is quick binary
  - only goes up to 9
  - anything bigger than 1001 is invalid
  - other codes exist too

computer architecture

assembly language

- hierarchy going down to hardware
  - interpreted language
    - e.g. python, ruby, Matlab
  - compiler languages
    - e.g. c, c++, pascal
  - assembly language
    - symbolic version of machine
  - machine code
    - set of permissible binary operations on the CPU
  - hardware
    - things happen here
- this has nothing to do with the operating system

code sample

- a small code sample `INIT MOVEQ.L #0, %D1`
  - `INIT` is a label
  - `MOVEQ` is a mnemonic
  - `.L` specifies the data size
  - `#0` is an immediate value
  - `%D1` is data register number 1
  - i.e. 32 bits containing the value 0 will be transferred to data register 1

some terminology

- assembling
  - conversion of assembly into machine
- disassembly
  - conversion of machine into assembly
- assembler
  - program that actually does the assembling
- hand assembly
  - assembling by hand
- resident assembler
  - runs on the same machine type as code
- cross assembler

  - assembler runs on a different machine type than the code
  - implies code must be transferred eventually
- compiler
  - program that translates higher level languages into machine code often using an assembler
  - object code
    * another term for machine language
    * may be joined with other machine code
  - linker
    * links many different object files together

assembly process

- going down again
  - text editor
    * like emacs and whatever
  - assembly
  - assembler
  - object code
  - linker
  - executable
- the object code and the linker are optional

machine language instructions

- e.g. `5342 <-> SUBQ.W #1, %D2`
  - `5342` is the machine code in hex
  - `SUBQ is` subtract quick
  - `.W` is specifying a data size of 16 bits
  - `#1` is the source or input
  - `%D1` is the destination or output
  - the entire example means to subtract 1 from D2 and store result in D2 $5342_{16} = 0101|001|101|000010$
  - the left most 4 bits encode `SUBQ`
  - the 3 bits after `SUBQ` denote the value of 1
  - the 3 bits after the value denote the data size
  - the last 6 bits specify the destination as D2
- all this is in the PRC
- the size of any instruction is in the PRC
  - there are always some extention memory addresses for any other information that may be used
- if the space where the PRC should list instruction size is empty then that instruction is illegal and cannot be done

- opcodes are the things written in that column
  - these specify what the instruction looks like in binary
  - the effective addressing mode is on the back
- e.g. `MOVE.L #0x20000, %D1`
  - the `#0x20000` is in immediate addressing mode
  - the opcode looks like `0010 RRR MMM eeeeee`
    * `RRR` = destination register = 001
    * `MMM` = destination mode = 000 (from effective addressing on back)
    * `eeeeee` = source effective address = `MMM RRR` of the source
      · in this case: `eee eee = MMM RRR = 111 100`
  - this is where that extension stuff from earlier comes into play
  - the actual value will be in the next 32 bits after the opcode
- the coldfire is a big endian machine
  - this means the least significant bits of a value are stored later

memory maps

- can be drawn in any fashion
- usually have a width as the number of bits
- the address number is on the side
- the coldfire is word oriented
- just an abstraction but helps give information about how the device stores data
- for instance, addresses may be assigned to each byte or 2 bytes
- look at jan 27

assembly instruction format

- e.g. `loop bra loop *terminate the program`
  - `loop` is a label
    * this allows statements to the address of this instruction
    * standard conventions
  - `bra` is a mnemonic
    * name of instruction to execute
    * usually has .L, .B, or whatever to specify data size
  - `loop` is an operand
    * listed in source, destination order
    * only for some instructions
  - `*term...` is a comment
    * can also be shown by `//` or `/* ... */` a la c

assembly directives

- not all lines contain cpu instructions
- some are assembler directives not executed by the cpu
- these are used by the assembler
- e.g.
  - `section <name>`
    - starts a new relocatable section of code
  - `end`
    - indicates to assembler that things are done
  - `include <fname>`
    - places contents of fname at the include location
  - `org <address>`
    - sets the assembler address counter and tells it where to put the code
  - whole bunch more

MCF5271 programmers' model

- data registers
  - accessed by `.B, .W, .L`
- general purpose address registers
  - accessed by `.W, .L`
  - funny thing about `.W`
    - any time a `.W` is moved into the address registers, it is sign extended
- special purpose address register
  - discussed later
- SR
  - the CCR or the condition code register is that last column in the PRC

data registers

- can opt to only use a portion of the whole register

address registers

- no .B access
- remember that things are sign extended when moved into the address registers

```
MOVEA.W #0xFEED, %A3
---
A3 = [FF FF FE ED]
```

Move word

program counter

- bit register
- always points to next instruction to execute
- controls program flow
- does not point to extensions

status register

- right now we're only interested in the lowest byte called the CCR as already mentioned. It's the condition code register. `CCR = [- - - X N Z V C]`
  - `-` are useless
  - `X` is extend
    - used for adding more than 32 bit long words
  - `N` is negative
    - weird thing about this is that it treats everything as signed
  - `Z` is zero
  - `V` is signed overflow
  - `C` is unsigned carry
- the condition codes are in the last column of the PRC
  - e.g. `- * * 0 0`
    - `X` doesn't change
    - `N` will be updated
    - `Z` will be updated
    - `V` will be 0
    - `C` will be 0

data alignment

- just the way the data might be arranged
- if the data is altogether in one memory location, it is aligned
- if it was spread over two memory locations, it is unaligned
- aligned data is faster to read
- coldfire has no alignment requirement but some cpus do
- look to jan 31

MCF5271 instruction set overview

- one categorization has 4 groups
  - data movement
  - sequence control
  - arithmetic and logic
  - special/control: not covered yet

data movement instructions

- `MOVE` actually copies
- coldfire cannot move from memory to memory. it must have a register in between
- `MOVEA` sign extends the `.W`
- `MOVEQ` is used for 8 bit values into a data register
  - this is sign extended
- `MOVE` CCR can do stuff to the CCR
- `MOVEM` can move multiple but not discussed yet

squence control instructors

- programs are not usually linear
- unconditional sequence control
  - change the location of the PC
  - e.g.
    * `BRA`: branch
    * `JMP`: jump
    * when writing assembly, the programmer will write `BRA <destination label>`
  - there's this weird branch relative addressing on the last page of notes
- conditional sequence control
  - Bcc instructions
  - change the CCR
  - unsigned
    * `BHI` - higher
    * `BHS` - higher or same
    * `BLS` - lower or same
    * `BLO` - lower
  - signed
    * `BGT` - greater
    * `BGE` - greater or equal
    * `BLE` - less or equal
    * `BLT` - less
  - both
    * `BEQ` - equal
    * `BNE` - not equal
  - these use the CCR to find out when to branch
    * `CMP` - data registers
    * `CMPA` - address registers
    * `CMPI` - immediate value
  - note that the C bit in the CCR is set by dest-src

        * `TST dest`
            · sets the CCR's N,Z values according to dest
            · does not change anything in dest
    - `BTST #bit, dest`
        * sets the Z bit in the CCR according the bit of dest

math instructions

- adding
    - `ADD` - src+dest->dest
    - `ADDI` - src is immediate
    - `ADDQ` - src is 0-7 immediate
    - `ADDA` - dest is an address register
    - `ADDX` - src+dest+X(CCR) -> dest
- subtracting
    - `SUB`
    - `SUBI`
    - `SUBQ`
    - `SUBA`
    - `SUBX`
- multiplication
    - `.W` - overflow not possible
    - `.L` - overflow possible
    - `MULU` - unsigned
    - `MULS` - signed
- division
    - `.W` - stores 16bit remainder and quotient
    - `.L` - stores 32bit quotient
    - `DIVU` - unsigned
    - `DIVS` - signed
- shifts
    - `ASL #bits, %DX`
        * shifts `#bits` to left of DX
        * MSBit goes to X,C
        * LSBit gets 0
    - `ASR #bits, %DX`
        * shifts right
        * MSBit gets copied into new MSBit
        * LSBit goes to X,C
        * the sign of the number is kept this way

- misc
  - `NEG %DX`
    * sign of DX gets switched
  - `NEGX %DX`
    * DX-X -> DX
  - `EXT`
    * does something
  - `CLR %DX`
    * clears the register (zeros)

logic instructions

- `AND <mask>, <dest>`
  - `<mask>` has 1s and 0s
  - all the bits in dest that correspond with the zeros in mask are cleared
- `ANDI <mask>, <dest>`
- `OR <mask>, <dest>`
  - similar to AND
  - all the bits in dest that correspond with the ones in mask are kept
- `ORI <mask>, <dest>`
- `EOR <mask>, <dest>`
  - similar to AND
  - all the bits in dest that correspond with the ones in mask are kept
- `BCLR #bit, <dest>`
  - clear bit to zero
- `BSET #bit, <dest>`
  - set bit to one
- `BCHG #bit, <dest>`
  - toggles bit
- shifting
  - `LSR`
  - `LSL`
  - similar to ASR and ASL but the sign is not retained in LSR

addressing modes

- `<ea>` is the effective address of any instruction

different modes

- immediate `ANDI.L #0xFF, %D7`
  - the `#0xFF` is encoded in the instruction itself so the `<ea>` = the address of the instruction
- absolute `CLR.B var1`
  - var1 is the effective address `CLR.B 0x1000`
  - `0x1000` is the effective address
- data register direct `CLR.B %D0`
  - `<ea>` is the data register
- address register direct `MOVEA.L #0x32, %A1`
  - `<ea>` is the address register
- register indirect `MOVE.B (%A1), %D0`
  - the `<ea>` is just whatever is inside A1
- register indirect with post increment `MOVE.L (%A2)+, %D3`
  - the `<ea>` is A2 before the instruction + 4
  - the 4 is because it is a .L
- register indirect with pre decrement `MOVE.B -(%A3), %D4`
  - the `<ea>` is the A3 before the instruction - 1
  - the 1 is because it is a .B
- register indirect with displacement/offset `MOVE.B 10(%A0), %D1`
  - the `<ea>` is A0 before the instruction + 10
- indexed register indirect with offset `MOVE.L -1(%A0, %D2*2), %D6`
  - the `<ea>` is `A0 + (D2 * 2) - 1`
  - the *2 is the scale factor
  - the SF can be 1,2,4,8 (only if floating point unit)
- PC relative with offset
  - the `<ea>` of the `Table(%PC)` is just Table
  - this instruction automatically calculates the offset from the PC. allows you to move Table around
  - this way is used because it allows the code to be be moved easily
- PC relative with index and offset
  - `<ea>` = String + D1
- special instructions
  - `LEA src, %AX`
    - load effective address
    - places the effective address of src in AX

- `PEA src`
    - pushes the `<ea>` to the stack

```
MOVE.W Table(%PC), %D1
Table: .word 0x1234, 0x5678

MOVE.L #2, %D1
MOVE.B String (%PC, %D1), %D0
String: .asciz "We rock"
```

subroutines

- allow modularity
- kind of like functions

stack

- the stack
    - used for saving return addresses for JSR and RTS
    - a stack is just a group of memory locations
    - Motorola devices have stacks that grow towards M(0)
    - adding items by pushing
    - removing items by pulling
    - we use A7 as the stack pointer
    - the pre decrement addressing mode is used with the SP for pushing stuff
    - the post increment addressing mode is used with the SP for pulling stuff

basic instructions

- there's a few common patterns in using the stack `MOVE.W #0x0227, -(%SP)`
    - decreases the SP by 2 and then places the value there `MOVE.B (%SP)+, 0x20000`
    - gets a byte from SP and places it at `M(0x20000)` and then increments the SP by 1
    - automatic stack use
        - `PEA` - discussed above
        - `JSR` - jump to subroutine
        - `BSR` - branch to subroutine
        - `RTS` - return from subroutine
        - the `JSR` and `BSR` push a 4byte address to the stack
        - the `RTS` pulls the 4byte address from the stack and puts it in the PC

stack frames

- stack frames simplify using the stack inside subroutines `LINK %AX, #n`
  - creates a stack frame
  - the `#n` is always negative
  - order:
    * push AX onto the stack
    * copy the SP to AX
    * add the immediate data to the SP `UNLK`
    * removes a stack frame
    * order:
      · copy the AX to the SP
      · pulls 4byte off the stack into AX
  - the `MOVEM` instruction is often used after the `LINK`

parameter passing

- three different ways:
  - CPU registers
    * input parameters are passed in through the registers
  - output is also through the registers
  - stack and cleaned in calling scope
    * input parameters are placed on the stack including (maybe) space for return value from the calling scope
    * subroutine grabs input data from the stack and places output in the appropriate place
    * calling scope then pulls from the stack and cleans the whole thing up
  - stack and cleaned inside subroutine
    * this one is weird
    * input parameters are passed in like normal but the stack is cleaned inside the subroutine
    * basically, say we pass in arg1, arg2 on the stack +rsp->| arg1 | | arg2 |r+
    * after the subroutine is done, the stack will look like this: `sp->| returnvalue|`
    * this one seems tedious
- prof recommends the stack and clean in calling scope

interfacing

parallel bus connected devices

- devices are connected using the address and the data bus
- these devices are said to be memory mapped devices
- examples: A/D converters, D/A converters, digital I/O, temp sensors, etc.

on chip peripherals

- these are peripherals actually on the same chip as the MCU
- these are very important to consider before purchasing a MCU

serial communication peripherals

- UART
  - asynchronous - clock is not transmitted
  - this means that the receiver and transmitter must be preconfigured to the same clock
  - LSBit is transmitted first
  - example: 9600 baud, 8E1
    * baud = 9600 bits per second
    * = # of data bits
    * E = even (parity)
    * = # of stop bits
- SPI
  - synchronous - master generates the clock
  - MSBit is transmitted first
  - SPI contains a ring of shift registers
  - basically, the slave shifts the MSBit out and it goes straight into the master
- I2C
  - synchronous - master generates the clock
  - more complicated than SPI
  - there is a single bidirectional data line called the SOA or SDA
  - the clock line is called the SCL
  - also includes an address line to assign addresses to each device from 0 to 7

general purpose I/O

- this is bascically the simplest peripherals that rely on simple on/off input
- open is usually 1 and close is 0
- modern MCUs allow almost any pin to be used as a digital I/O
- data direction register
  - keeps track of whether registers are set to input or output
- output/input data register
  - each bit sets whether the output is high or low
  - or each bit sets whether the input is high or low
  - each bit corresponds to a particular pin
- we can emulate peripherals by just controlling the pins with software

voltage, current, and timing

- these need to be checked for successful interfacing
- pin voltages/currents are usually in the DC characteristics section of the data sheet
- timing is usually found in the AC characteristics
- basically we need to ensure that the correct current, voltage is going through the peripheral
- we also need to check if the clock is working according to what the peripheral requires

exceptions

interrupts

- generated with hardware
- interrupts are a form of exceptions
- interrupts are an alternative to polling
- we basically wait for the device to ping us rather than asking the device constantly
- traps
  - generated internally by the CPU
- enabling and priorities
  - CPU needs to be configured to listen for the interrupt
  - we can do this with masks
  - if something is masked, we can't see it
  - most CPUs have two levels of masking
  - the coldfire uses the system bits of the status register as mask bits
  - I2, I1, I0 are the inhibit mask
  - `anything <= (I2, I1, I0)` is inhibited
  - some interrupts are non-maskable
  - these interrupt priorities are set by:
    * handwriting
    * hardware
    * firmware
  - note that higher priorities are more important

vector table

- there are three ways for a CPU to figure out what to do after an exception occurs
  - user interrupts
    * complicated peripheral gives info about routine address
    * coldfire
  - autovectored interrupts
    * hardware contains addresses of routines for interrupts somewhere

    * the CPU just finds the appropriate one
  - simple
    * some simpler devices just go to the same address no matter what exception occurs
  - example: if a divide by zero exception occurs, the CPU goes to the vector table and gets the address of the routine from the hardwired location in the table

exception processing

- process
  - exception occurs
  - current instruction is completed
  - CPU context is saved
    * PC is pushed to stack
    * push the status register to stack
    * coldfire has another word
  - the status registers, I bits are set to the exception's priority hiding anything that's less important
  - the interrupt service routine (ISR) address is read from the vector table and placed in the PC
  - ISR executes
  - somewhere in the ISR, the exception is cleared somehow
  - the ISR has a return from exception (rte) at the end similar to rts for subroutines
  - PC and SR are pulled from the stack
  - rest of the program continues to execute
- ISRs should be short and executed quickly
- nested interrupts
  - sometimes the ISR generates an exception itself
  - this then only gets called if it is a higher priority than the original exception
- example
  - coldfire has a trap instruction: `TRAP #n , n=0,...,15`
  - this generates an exception and executes code for the appropriate trap instruction from the vector table
  - we can rewrite a trap instruction so that it swaps two registers +rTrap: move.l %d0, -(%sp) move.l %d1, %d0 move.l (%sp)+, %d1 rter+
  - we can draw a memory map at the beginning of this Trap
  - the memory map is 16-bits wide +r->SP| mystery word | | status register | | returned 2MSBytes | | returned 2LSBytes |r+
  - now we actually need to place the address of this code in the vector table `move.l #Trap, VECTORBASE+0x80+4*2`
    * `VECTORBASE` is the address of the vector table
    * `0x80` is the start of the trap instructions
    * *2 is the offset for `Trap #2`
  - to actually call this interrupt we write: +rmove.l #0xFFFF FFFF, %d0 move.l #0xABBA 0000, %d1 TRAP #2 /* the two registers are now swapped */r+