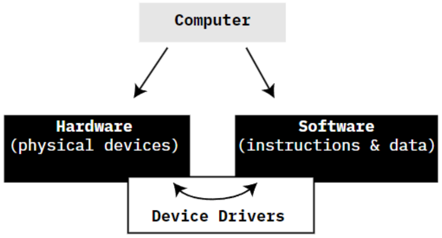
CSC 230 Cumulative Notes Summary (for finals)

Introduction to Computer Architecture (University of Victoria)

Components of a Computer System



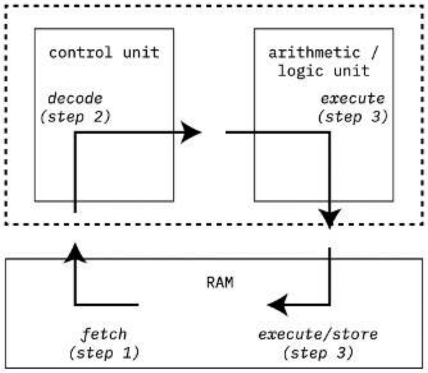
Hardware and software “communicate” through pieces of specialized code known as “device drivers”.

Computer Hardware

- **Devices**: e.g. CPU, Memory, Display controller, Keyboard controller, Touchpad controller, SSD controller

- **Bus**: the multiple-wire communication system that transfers data between component *devices* inside a computer, effectively connecting devices together

- **Processors**: run on the fetch-decode-execute cycle
- **Fetch-decode-execute cycle**:
  - Fetch machine instruction from program memory (RAM)
  - Instruction decoded to derive meaning
  - Actions indicated from decoding is executed
- **Clock pulses**: each pulse is the start of a fetch-decode-execute cycle; goes on forever; cycle’s frequency is the **processor speed**

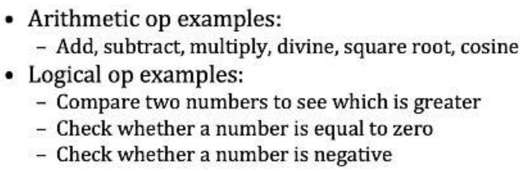


CPU (processor)

→ the coordinator of every computer system

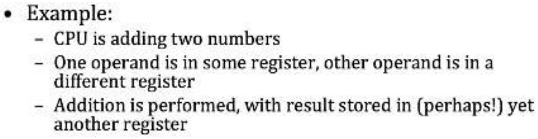
3 main parts:

- **Control Unit** (CU)
  - Decides what CPU components are to be used next
  - Makes sure everything in CPU connected for specific instruction
- **Arithmetic/Logic Unit** (ALU)



- Arithmetic op examples:
  - Add, subtract, multiply, divine, square root, cosine
- Logical op examples:
  - Compare two numbers to see which is greater
  - Check whether a number is equal to zero
  - Check whether a number is negative

- **Registers**
  - Memory-like locations in CPU
  - Hold operands used by current ALU operations, & holds results of those operations



- Example:
  - CPU is adding two numbers
  - One operand is in some register, other operand is in a different register
  - Addition is performed, with result stored in (perhaps!) yet another register

Memory

→ contains code of running program & contains data (variable, constants) for running program

Von Neumann Architecture

- Same memory is used for code and data

Harvard Architecture

- Code is in one memory system
- Data is in a separate, diff memory system
- Fetch-decode-execute cycle in Harvard:
  - Two cycles are req to execute one instruction: fetch&decode, execute
  - BUT the next instruction is fetch&decode while the previous is being executed

System Bus Model

Address bus

- Controlled by CPU - when CPU interested in retrieving values in memory

Data bus

- Contain contents of memory or results of input which we might want to load through CPU
- Carries results CPU wants to be stored in memory

Control bus

- Indicates there is some read or write occurring
- Deals with contention - devices speaking directly to the memory without involving CPU

Bases

Decimal (10)

- No prefix or suffix

Binary (2)

- Prefix: 0b , %
- Suffix: b

Octal (8)

- Prefix: 0o , 0
- Suffix: o

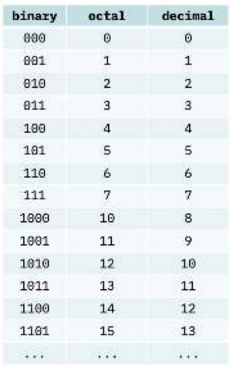
Hexadecimal (16)

- Prefix: 0x or $
- Suffix: h

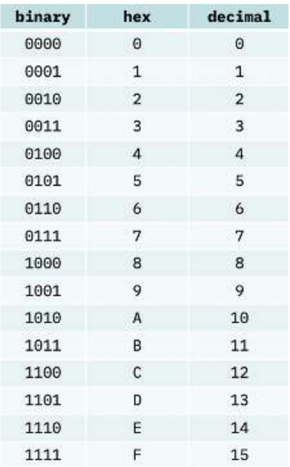
Conversions

Binary, Octal, Hexadecimal → Decimal

1. Multiply the digit by the base (2, 8, 16) to the power of the *nth* position the digit is located in the number (see: Horner's Algo.)
   a. *nth* pos. starts from 0 on right



binary	octal	decimal
000	0	0
001	1	1
010	2	2
011	3	3
100	4	4
101	5	5
110	6	6
111	7	7
1000	10	8
1001	11	9
1010	12	10
1011	13	11
1100	14	12
1101	15	13
...	...	...



binary	hex	decimal
0000	0	0
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

2. Sum all the values

Octal, Hex → Binary

- Convert to decimal, then binary

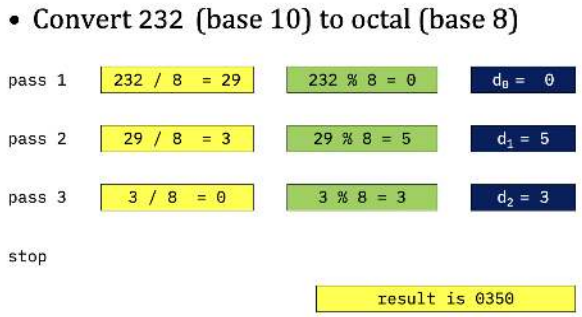
Octal ↔ Hex

1. Convert to decimal, then binary
2. Count out binary bits from right
   a. To Octal: groups of 3 bits
   b. To Hex: groups of nibbles (4)
3. Rewrite groups into appropriate digits

Decimal → Binary, Octal, Hexadecimal

- digits are produced from right to left

Repeated division algorithm:



- Convert 232 (base 10) to octal (base 8)

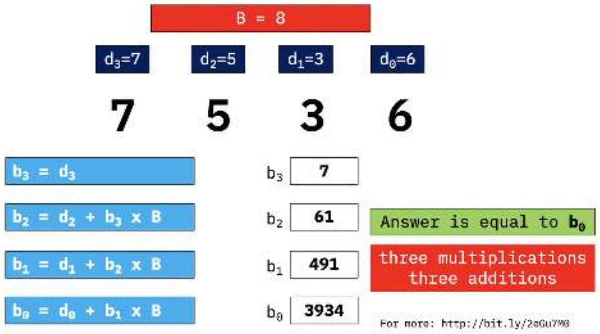
pass 1	232 / 8 = 29	232 % 8 = 0	d_0 = 0
pass 2	29 / 8 = 3	29 % 8 = 5	d_1 = 5
pass 3	3 / 8 = 0	3 % 8 = 3	d_2 = 3
stop			

result is 0350

Horner's Algorithm

→ converting binary/octal/hex to decimal can be expensive b/c the multiplications increases runtime in program

Solution - reduce number of multiplications:



Positive Integer ↔ Negative Integer

1. Find binary representation of integer
2. Use *change-sign rule* to convert into ***two's complement*** (also works from neg to pos)
   a. If the original number has 0 as leftmost bit (is positive), 1 is neg
   b. Flip all bits to the left of the rightmost (bit) set (1)

Binary-number Terminology (MIPS)

- **Bit**: binary digit
- **Nibble**: 4 bits
- **Byte**: 8 bits
- **Word**: 16 bits (for AVR)
- **Double word**: 32 bits (for AVR)

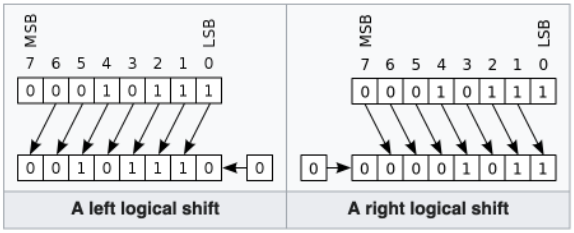
- least-significant bit = right-most bit = bit 0
- most-significant bit = left-most = # bits - 1
- set = 1 = true
- un-set = 0 = false

Min & Max bit representations

Given *k* = # of bits the binary number has

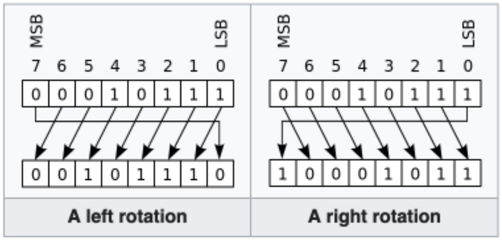
Unsigned Min	Unsigned Max	2's complement Min	2's complement Max
0	$2^k - 1$	-2^{k-1}	$2^{k-1} - 1$

Bit Shifts



- Left shift == byte * 2
- Right shift == byte / 2

Bit Rotate



Boolean Operations

NOT - complement ; *a'*

- Inverts the boolean value of a bit

a	NOT a
0	1
1	0

AND - conjunction ; *ab*

- True only if both bits are 1
- Is the carry for bit addition
- Used in set intersection

a	b	a AND b
0	0	0
0	1	0
1	0	0
1	1	1

OR - disjunction ; *a+b*

- True if one operand is true
- Used in set unions

a	b	a OR b
0	0	0
0	1	1
1	0	1
1	1	1

XOR ; $a \oplus b$

- True only if exactly one operand is true
- Is the truth table for binary addition

a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

NAND, NOR, XNOR

- The opposite of AND, OR, XOR

a	b	a NAND b
0	0	1
0	1	1
1	0	1
1	1	0

a	b	a NOR b
0	0	1
0	1	0
1	0	0
1	1	0

a	b	a XNOR b
0	0	1
0	1	0
1	0	0
1	1	1

Byte Order “endian-ness”

→ convention on how bytes are stored in memory

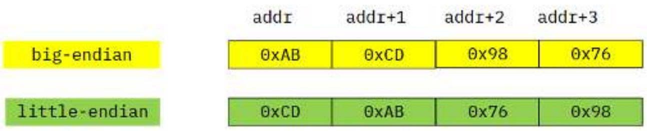
Big-endian:

- Most significant byte of a word in addr, least significant in addr + 1

Little-endian:

- Most significant byte in addr + 1, least significant byte in addr

E.g. given: `0xABCD` and `0x9876`



	addr	addr+1	addr+2	addr+3
big-endian	0xAB	0xCD	0x98	0x76
little-endian	0xCD	0xAB	0x76	0x98

- Machine codes (opcodes) in memory are stored as little-endian; we input it in big endian

AVR Arithmetic

Binary Addition

Half-adder: takes two incoming bits A & B

- Sum = A XOR B
- Carry = A AND B

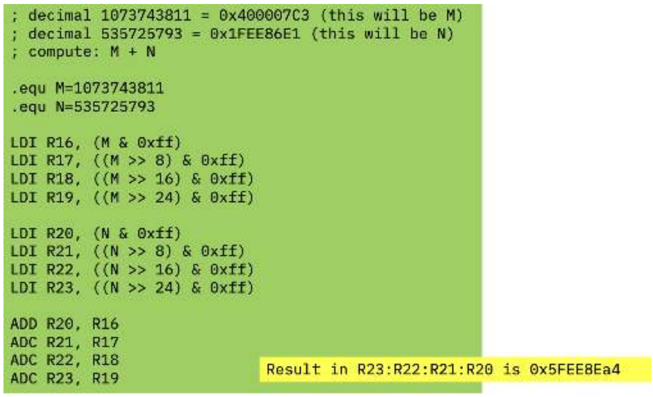
Full-adder: takes three bits A, B, and carry bit C

- Sum = A XOR B XOR C
- Carry = (A AND B) OR ((A XOR B) AND C)

* “positive” and “negative” integers in AVR are from our own interpretation == must choose unsigned or signed for a solution and stick with the choice throughout the solution’s code

* to add/subtract with values beyond 0 < x < 255 or -128 < x < 127, must write our own code to do operations with more than two pairs of bytes

= ADD low bytes together, ADC high bytes



```
; decimal 1073743811 = 0x400007C3 (this will be M)
; decimal 535725793 = 0x1FEE86E1 (this will be N)
; compute: M + N

.equ M=1073743811
.equ N=535725793

LDI R16, (M & 0xff)
LDI R17, ((M >> 8) & 0xff)
LDI R18, ((M >> 16) & 0xff)
LDI R19, ((M >> 24) & 0xff)

LDI R20, (N & 0xff)
LDI R21, ((N >> 8) & 0xff)
LDI R22, ((N >> 16) & 0xff)
LDI R23, ((N >> 24) & 0xff)

ADD R20, R16
ADC R21, R17
ADC R22, R18
ADC R23, R19
```

Result in R23:R22:R21:R20 is 0x5FEE8Ea4

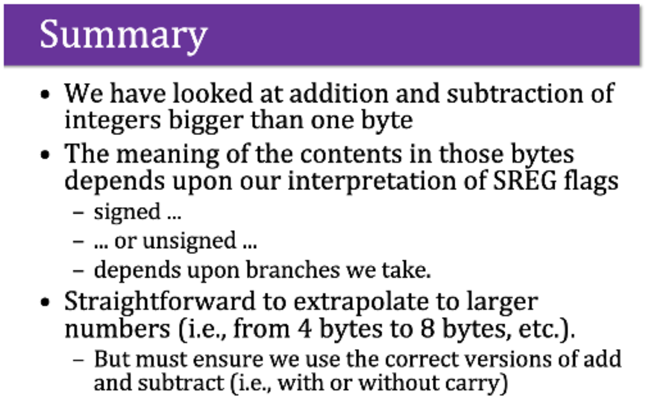
Binary Subtraction

* use two-byte signed integers

= the bytes will represent positive values in the range of -32768 to 32767

* similar to add: SUB low bytes, SBC high

- the C helps in 15-20 situations
- activation of C flag lets high byte operation know that a borrow is needed



Summary

- We have looked at addition and subtraction of integers bigger than one byte
- The meaning of the contents in those bytes depends upon our interpretation of SREG flags
  - signed ...
  - ... or unsigned ...
  - depends upon branches we take.
- Straightforward to extrapolate to larger numbers (i.e., from 4 bytes to 8 bytes, etc.).
  - But must ensure we use the correct versions of add and subtract (i.e., with or without carry)

Functions in AVR

if (a >= b)

a < b : BRLO

if (a > b)

a <= b : BRSH

if (a >= b)

a < b : BRLT

if (m == 0)

if z flag : BRNE

RJMP vs. JMP

RJMP:

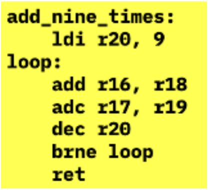
- Encoded opcode is 2 bytes in size
- Execution requires 2 cycles
- Requires less CPU and memory
  - Good when memory is limited

JMP:

- Encoded opcode is 4 bytes in size
- Execution requires 4 cycles

Looping something set # of times:

- Place loop into function
- **RET** at end of loop > goes back to beginning of function



```
add_nine_times:
    ldi r20, 9
loop:
    add r16, r18
    adc r17, r19
    dec r20
    brne loop
    ret
```

CALL & RET

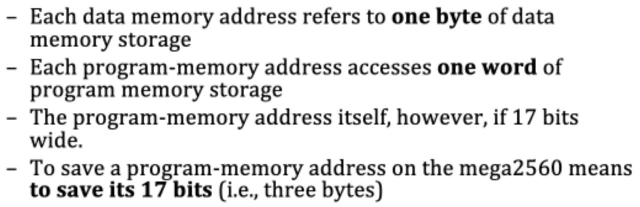
> they modify the PC & therefore the normal sequential flow of control
- RET overwrites the PC w/ most recent values saved on the stack (in internal SRAM); CALL writes
- their stack operations are implicit

Stack

> is in Internal SRAM
- as values are pushed onto the stack, the stack grows towards low memory (0x0200)
- values popped, stack shrinks towards high memory (0x21ff)

- **Stack pointer** register is stored in the I/O register area

Memory addresses



- Each data memory address refers to **one byte** of data memory storage
- Each program-memory address accesses **one word** of program memory storage
- The program-memory address itself, however, if 17 bits wide.
- To save a program-memory address on the mega2560 means **to save its 17 bits** (i.e., three bytes)

Using PUSH & POP in code

- PUSH: copies the contents of a register onto stack
  - register's value is now on the top of the stack
  - stack size is increased by one
  (SP is decremented by 1)

- POP: copies the contents top of stack onto register
  - stack's size decreased by 1
  (SP is incremented by 1)

*stack grows downwards & shrinks upwards
* stack should be set up at the start of program:

- By convention placed at the top of RAM, but can be stored in some other location in SRAM

Uses:

- Store one-byte intermediate values
- Temporarily store a register's value by pushing onto the stack
  - Re-use the register for something else
  - Once done with register, restore its value by popping the value off the stack and back into the register

Return values w/ Function

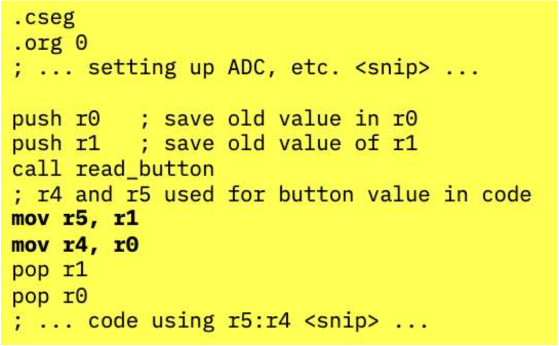
Two parts:

- Code expects a return value from calling a function (invoke using CALL)
- Code in the function performs the computation that produces the return value

Two main mechanisms:

1. Return value stored in designated general-purpose register
   - Must save a register's value within another "safer" register
   - OR push the value onto the stack before the call, and pop it back off after the return value has been used

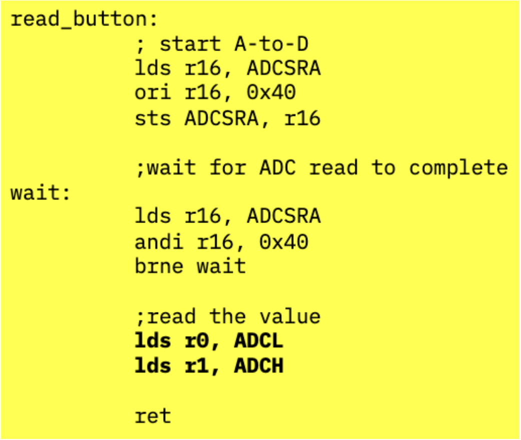
calling:



```
.cseg
.org 0
; ... setting up ADC, etc. <snip> ...

push r0   ; save old value in r0
push r1   ; save old value of r1
call read_button
; r4 and r5 used for button value in code
mov r5, r1
mov r4, r0
pop r1
pop r0
; ... code using r5:r4 <snip> ...
```

called:



```
read_button:
          ; start A-to-D
          lds r16, ADCSRA
          ori r16, 0x40
          sts ADCSRA, r16

          ;wait for ADC read to complete
wait:
          lds r16, ADCSRA
          andi r16, 0x40
          brne wait

          ;read the value
          lds r0, ADCL
          lds r1, ADCH

          ret
```

2. Return value placed on stack

- not a currently good approach because have to save many values in registers

Function Parameter

Parameters placed in registers

call-by-value: caller's argument (i.e. it's value) is copied into callee's parameter; parameters contain the values of program's quantities

call-by-reference: memory location of the caller's argument is used as the callee's parameter; parameters contain addresses of memory locations, which contain value

- Able to provide a parameter located in any memory location in SRAM
- Data memory indicated can be large..

actual parameter: passing an argument to a method in java - what the code works with

formal parameter: the informative, placeholder parameter that is specified in method construction in java

Parameters placed on stacks

- Combine pushes/pops with memory loads
- Some operations performed by caller
- Some operations performed by callee

Stack frame:

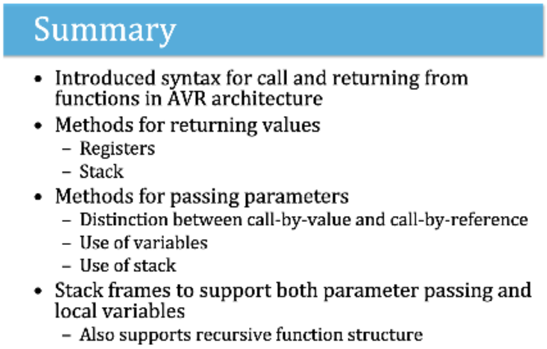
> a region of stack that is specific to the callee's parameters and callee's local data

- callee code will treat the stack as a block of memory, accessible using LDD & offset

local variables/local storage:

> set aside room on the stack for local storage by subtracting the needed number of bytes from the stack pointer

- ADIW (remove storage) - in epilogue
- SBIW (add storage) - in prologue
    - Meant to be used to w/ X, Y, Z

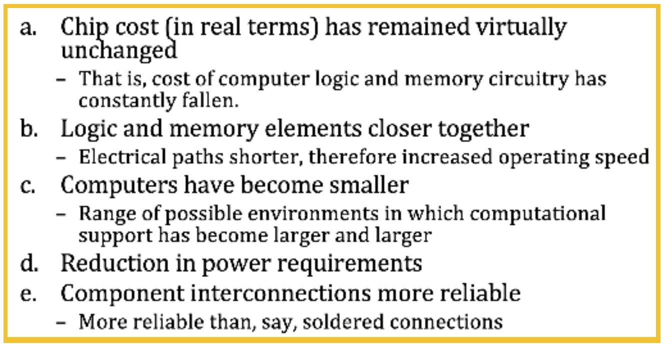


Summary

- Introduced syntax for call and returning from functions in AVR architecture
- Methods for returning values
    - Registers
    - Stack
- Methods for passing parameters
    - Distinction between call-by-value and call-by-reference
    - Use of variables
    - Use of stack
- Stack frames to support both parameter passing and local variables
    - Also supports recursive function structure

Performance Issues

Moore's law: prediction that number of transistors on a chip would double every 1.5 years; consequences:

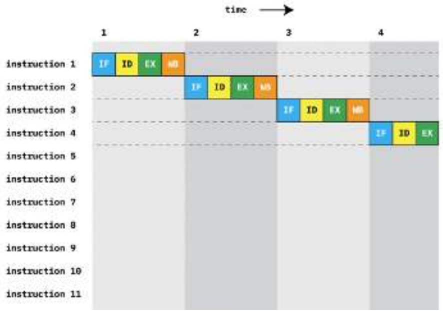


a. Chip cost (in real terms) has remained virtually unchanged
   - That is, cost of computer logic and memory circuitry has constantly fallen.
b. Logic and memory elements closer together
   - Electrical paths shorter, therefore increased operating speed
c. Computers have become smaller
   - Range of possible environments in which computational support has become larger and larger
d. Reduction in power requirements
e. Component interconnections more reliable
   - More reliable than, say, soldered connections

Techniques to increase microprocessor speed

- Pipelining
- Branch prediction
- Superscalar execution
- Data flow analysis
- Speculative execution

Before pipelining

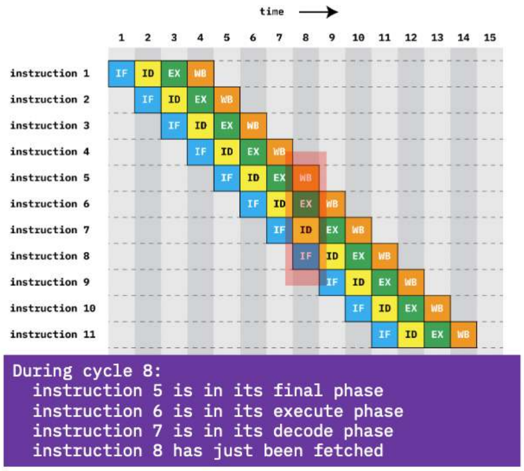


4 instruction phases in one clock cycle:

- Instruction fetch
- Instruction decode
- Execution
- Writeback

Pipelining

- Shrinks cycle time; increases clock frequency
- Diff instructions at diff stages can be worked on simultaneously
    - "Instruction-level parallelism"

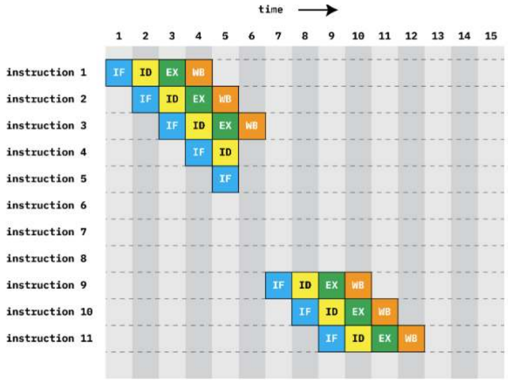


During cycle 8:
instruction 5 is in its final phase
instruction 6 is in its execute phase
instruction 7 is in its decode phase
instruction 8 has just been fetched

Branching:

> if a branch is taken, then the work for the instructions still in the pipeline is discarded

e.g. instruction 3 was a branch:



Pipeline Summary

> **goal**: to improve performance by increasing instruction throughput

> **how**:

- processing multiple instructions in parallel
- each instruction is at a different phase of execution
- each instruction has the same latency

> **hazards**:

- **Control hazard**: see branching
- **Structural hazard**: more than one instruction in the pipeline needs the CPU resource in the same cycle
- **Data hazard**: instruction in earlier stage requires data that will be the result of a instruction in later cycle (i.e. results only available after writeback phase)

Branch prediction

Idea: use actual program behavior to predict future behavior

- Processor keeps track of branch history
- When the CPU reaches a branch instruction, it fetches the state of the branch
- If the state is "branch normally taken" = target address fetched

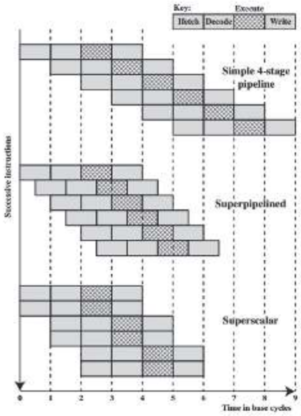
= if the branch is correctly predicted, :)

BUT prediction can be wrong sometimes, = extra work

= work has been done on increasing prediction accuracy (like using more history to predict)

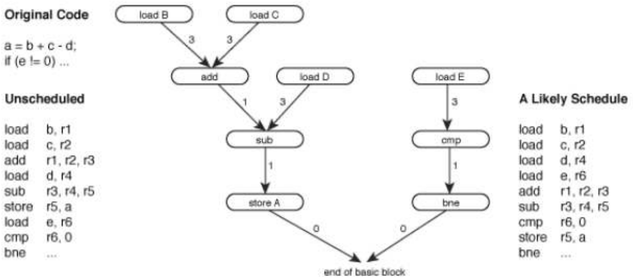
Superscalar execution

> multiple parallel pipelines



Data flow analysis

- Processor determines possible equivalent orderings of instructions:
  - **Goal**: choose an order that executes the fastest
- **Data flow**: analyzing code to discover how the results of computations are used as inputs/operands for later computations



Original Code

```
a = b + c - d;
if (e != 0) ...
```

Unscheduled

```
load   b, r1
load   c, r2
add    r1, r2, r3
load   d, r4
sub    r3, r4, r5
store  r5, a
load   e, r6
cmp    r6, 0
bne    ...
```

A Likely Schedule

```
load   b, r1
load   c, r2
load   d, r4
load   e, r6
add    r1, r2, r3
sub    r3, r4, r5
cmp    r6, 0
store  r5, a
bne    ...
```

Speculative execution

> combination of branch prediction & data-flow analysis

- Mainline code and speculative code can be evaluated in parallel

Techniques to increase transfer speeds between CPU & DRAM

Increase bit width

- More data transferred per DRAM read/write

Change DRAM interface

- Add more buffering/ caching onto DRAM chips & subsystem itself

Cache more on CPU

- Add more buffering/caching on CPU/ processor itself

Change bus structure

- Change interconnect structure, hierarchy of buses, in order to increase transfer speeds

Other Techniques to increasing speed

Multicore

> multiple cpu

- Multithreaded code makes use of multiple cores

MIC (many integrated core)

> more cores, w/ each core running slower than non-MIC

- Slower cores will experience less memory latency

GPUs

> specialized cores for video processing, etc.

Calculations

To find overall effective memory access time:

level one cache

(T1 * L1 hit ratio) + [(Tm + T1) * (1 - L1 hit ratio)]

> hit + miss

level two cache

(T1 * L1 hit ratio) + [(T2 + T1) * L2 hit ratio] + [(Tm + T1 + T2) * (1 - L1 hit ratio - L2 hit ratio)]

To find new running time:

$$T(1 - f) + \frac{T \times f}{N}$$

f = "fraction"

To find speedup:

$$\frac{T}{new\ running\ time} = \frac{1}{(1-f) + \frac{f}{N}}$$

To find time to execute a given program:

$$T = I_c \times CPI \times \tau$$

$$T = I_c \times [p + (m \times k)] \times \tau$$

Interrupts

polling

> some device external to the main CPU has a state that must be checked by the CPU for readiness

> at readiness, an action must be taken

> that state is checked, and actions are taken via code contained in an infinite loop

(!) more devices to manage means more to check = more elif clauses in loop

(-) even if no device is ready, the CPU is still going around in the loop, consuming CPU cycle

(-) mixing polling code with non-polling code difficult

interrupt

> alternative to polling; an event (signal) that can temporarily halt ("interrupt") the control flow of the currently running program in order to execute a task on behalf of that event (signal)

- Events triggered by:
  - External signals (ie. I/O devices)
  - Internal signals (ie. timer, CPU errors)
- The running program is temp. suspended while the task is completed
  - Task: (ie. interrupt handler)
  - CPU handles the transfer of control

Implicit vs. explicit

- Main code does not call the interrupt-handler function
  - CPU transfers control to handler
- Interrupt handler function explicitly causes control to return to interrupted code
  - Using RETI instead of RET

Overall roadmap of interrupt

1. With interrupts disabled, code that configures the interrupt handler mappings is executed
2. When configuration complete, interrupts enabled
3. Start main code execution
4. If event occurs that is associated with an enabled interrupt
    a. The CPU pauses where we are in the code
    b. CPU transfers control to the current handler for that event
    c. That handler disables interrupts
    d. Handler does its work (execute code within the interrupt handler)
    e. When handler is finished, it re-enable interrupts
    f. Handler causes CPU to return control to where we originally paused

Interrupt Handler

> should be the smallest amount of code necessary to deal with the interrupt

- For I/O devices: reading bytes from, or writing bytes to, the registers of the device
- For timers: setting status variables to important values

- Code within an interrupt handler very rarely calls other functions
    - Lack of time
    - Don't want to spend so much time on one interrupt that we miss another device's interrupt
- Handlers must ensure the interrupted code can safely resume operation after handler is done

Enabling Interrupts

Two levels:

1. Enabling the interrupts for the device itself "turning on the device"
2. Enabling the CPU to respond to interrupts in general
    - Is a single operation: SEI

NMI: vector for the RESET signal

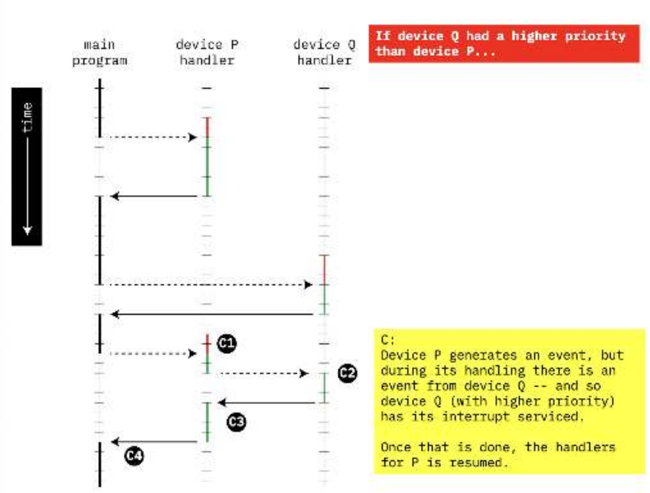
- An interrupt that can never be ignored

Multi-level interrupts

> ATmega2560 only support single-level interrupts

> multi-level interrupts have interrupt priority

- Devices with high-priority interrupts are permitted to interrupt the handlers for lower-priority devices



If device Q had a higher priority than device P...

C:
Device P generates an event, but during its handling there is an event from device Q -- and so device Q (with higher priority) has its interrupt serviced.

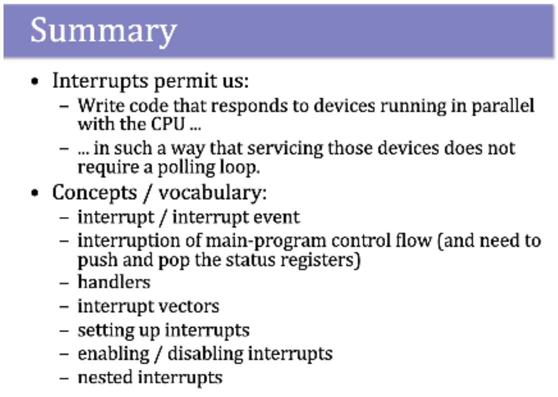
Once that is done, the handlers for P is resumed.

Critical section code

> when the code should not be interrupted

e.g. when OS is executing some sensitive code within the OS kernel itself

- CLI disables interrupts
- SEI enables interrupts



Summary

- Interrupts permit us:
    - Write code that responds to devices running in parallel with the CPU ...
    - ... in such a way that servicing those devices does not require a polling loop.
- Concepts / vocabulary:
    - interrupt / interrupt event
    - interruption of main-program control flow (and need to push and pop the status registers)
    - handlers
    - interrupt vectors
    - setting up interrupts
    - enabling / disabling interrupts
    - nested interrupts

Assembly Language Intro

Program memory: contains binary for opcodes

Instruction register: holds encoded instruction while it is being encoded

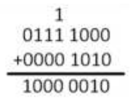
Instruction decode: holds instruction that has already been decoded (and can now control the ALU and datapaths)

Program counter (PC): holds/stores address of next instruction to be executed; starts at 0 and increments as it moves down program memory

SREG Flags:

H: half carry flag

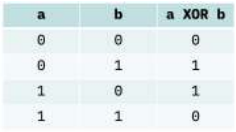
• if there is a carry moving from one nibble (4 bits) to the next, H flag set



```
   1
 0111 1000
+0000 1010
 1000 0010
```

S: sign flag

• the result of N flag XOR V flag



a	b	a XOR b
0	0	0
0	1	1
1	0	1
1	1	0

V: overflow flag

• if data is beyond this threshold: 127 < signed number < -128

‣ V flag is set to 1

N: negative flag

• indicates whether data is +/-

◦ if result has LMB as 1, N flag set to 1

Z: zero flag

• when adding/subtracting binary numbers

• if the result is 0, Z flag set to 1 (else is 0)

= 1 means the two values compared are equal in value

• branch opcodes: BREQ & BRNE check if the Z status flag is set or clear

• set/clear Z opcodes: CP & CPI

C: carry flag

• when adding binary numbers, if there is extra bit (carry bit), sets C flag to 1

Pseudo Registers/ pointers:

X: R26 (XL), R27 (XH)

Y: R28 (YL), R29 (YH)

Z: R30 (ZL), R31 (ZH)

• are used for read and writes from SRAM

◦ direct load and store: LDS, STS

◦ indirect load and store: LD, ST

- storing 0x77 in SRAM @ address 0x200

```
LDI R16, 0x77
STS 0x200, R16
```

```
LDI R16, 0x77
LDI R27, HIGH(0x200)
LDI R26, LOW(0x200)
ST X, R16
```

I/O Registers:

> registers to help us access ports

• IN, OUT: ports A to H

◦ channels; port register addresses

- Faster I/O
- Address must be known at runtime

• LTS, STS: all ports A to L

◦ data memory addresses

- More flexibility
- Address may be determined at runtime

◦ DDRx: data direction reg.

= used to control pin input/output

= 0 means input

= 1 means output

◦ PORTx: pin output reg.

= write 0 or 1 to a pin

◦ PINx: pin input reg.

= determine if 0 or 1 at specific pin

Directives

> give control to your program:

- To define constants
- To set aside memory for variable data
- To organize memory

Can:

• set aside addresses in SRAM

• define symbolic name for register (.def)

• equate symbol to a value or expression (.equ)

Expressions

- << shift left
- >> shift right
- & bitwise AND

*registers cannot be used in expressions

Memory

8 different dimensions to characterize memory

1. Location

- internal
- external

2. Capacity

- number of words available in memory device OR number of bytes

3. Unit of transfer

- internal memory:
  - Number of bits read out of/ written into memory at one time
- external memory:
  - Data transferred in units larger than a word (called blocks)

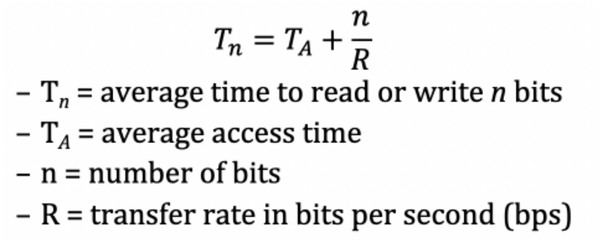
4. Access method

- sequential access
  - Data accessed in specific linear sequence
- random access
  - Each memory location can be accessed independent of others at no extra time & energy cost
- direct access
  - Location of memory block determined by address
  - Searching, counting, waiting used to reach final location
- associative
  - Word retrieved based on contents of same key

5. Performance

= 3 parameters used: access time, memory-cycle time, transfer rate

- access time (latency)
  - Random access: duration of time from the instant an address is presented to the instant the data is available for us
  - Other memory types: time to position read/write mechanisms at needed location
- memory cycle time
  - Access time + additional time for electrical transients to die out on signal lines
- transfer rate
  - Rate at which memory can be transferred into or out of a memory unit



$$T_n = T_A + \frac{n}{R}$$

- T_n = average time to read or write n bits
- T_A = average access time
- n = number of bits
- R = transfer rate in bits per second (bps)

*T_A can mean different things based on the mechanism that is acting:

- Hard drive:
  - Time to position arm over correct track, rotational latency for sector to arrive under read/write head
- Dynamic RAM:
  - Time from when the appropriate page of memory cells is first “strobed” to when bytes from that page can be read/written

6. Physical type

- Semiconductor memory: SRAM, DRAM, flash…

7. **Physical characteristics**

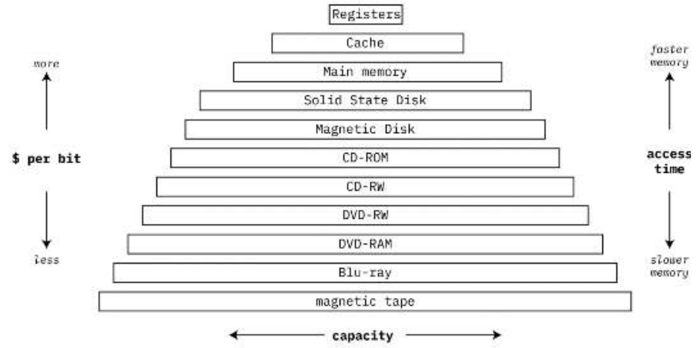
- *volatile* memory
  - Information that decays “naturally”
  - Is lost when electrical power is removed
- *non-volatile* memory
  - Once recorded, information remains without deterioration
  - A timescale needs to be indicated (it is not necessarily *archival*)
- semiconductor memory
  - Non-erasable version: read-only memory (ROM)
  - Re-writable version: EEPROM (electrically erasable programmable read-only memory)

8. **Organization**

- how are bits physically arranged to form words?
  - Banks, ranks, pages, NAND, NOR memory…
- other bits in addition to data bits?
  - Error detection?
  - Error correction?

Memory Hierarchy

> faster access time = greater cost per bit
> greater capacity = smaller cost per bit, but slower access time
∴ use multiple memory components

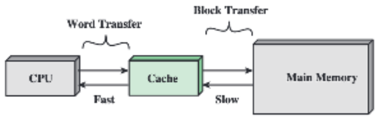


Locality of Reference

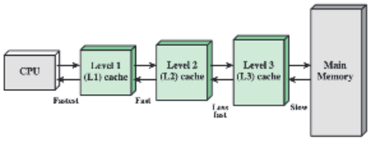
> memory references made by the processors (ie. reads and writes) are clustered in a small range of addresses rather than dispersed throughout the whole program
> over a short period of time, a processor tends to work only within a few clusters of addresses

How it helps with efficiency:
• changing clusters are moved to faster, level 1 mem
• old clusters are moved to slower, level 2 memory
• **hit ratio** describes the fraction of mem-refs satisfied by references to level 1== if ratio good, system performance good

Caching



A single cache



A multi-level cache (here: three levels)

• if an address has n bits, then number of words: 2^n
• M is the number of blocks in memory
  • $M = 2^n / K$
• a cache consists of blocks called lines
  • each line is K words in size

Tag: info to identify which main-memory block is in the line (increases the size of the line)

Cache Design Elements

Cache addresses
> can be physical or virtual/ logical addresses
- Logical: generated in virtual address space
- Physical: used to access real RAM

• memory management unit (MMU) converts logical addresses into physical, for currently running process

Cache size
> keep it small to keep costs low and processes fast, but large enough to keep hit ratio high

Mapping function
> how to determine which cache line should store a given memory block?
Three ways:
- Direct mapping (not flexible)
- Associative mapping (most flexible, expens.)
- Set-associative mapping (combo deal)

Direct mapping
> calculating which cache line the main memory should be:

$$i = j \% m$$

i = cache line number
j = main memory block number
m = number of lines in the cache
Cache line size = 2^w

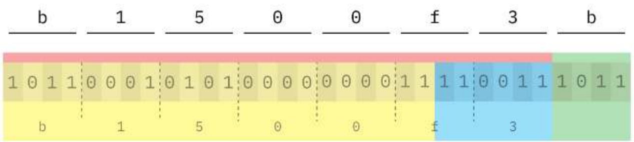
• size of offset (in bits): size of block = 2^{offset}
  from the right
• block: everything but the offset
• size of line (in bits): size of cache= 2^{line}
  starting from end of offset
• tag: everything after the offset and line

```
Size of address: 32 bits
Size of block: 16 bytes (2⁴ bytes)
Size of cache: 64 lines (2⁶ lines; 2¹⁰ bytes)

Example address: 0xb1500f3b
```

A change:

w is 4
m is 64

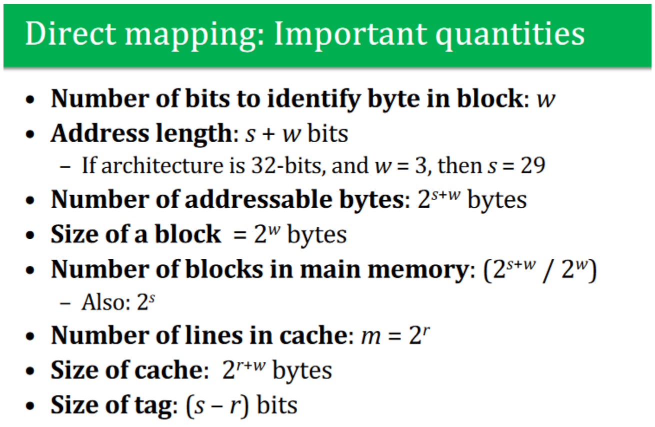


offset is 11 (0xb)

block is 0xb1500f3

line is 51

tag [illegible]



Direct mapping: Important quantities

- **Number of bits to identify byte in block**: w
- **Address length**: $s + w$ bits
  - If architecture is 32-bits, and $w = 3$, then $s = 29$
- **Number of addressable bytes**: 2^{s+w} bytes
- **Size of a block** $= 2^w$ bytes
- **Number of blocks in main memory**: $(2^{s+w} / 2^w)$
  - Also: 2^s
- **Number of lines in cache**: $m = 2^r$
- **Size of cache**: 2^{r+w} bytes
- **Size of tag**: $(s - r)$ bits

> what if the cache line to store a new memory address already contains a different block?
= Cache miss

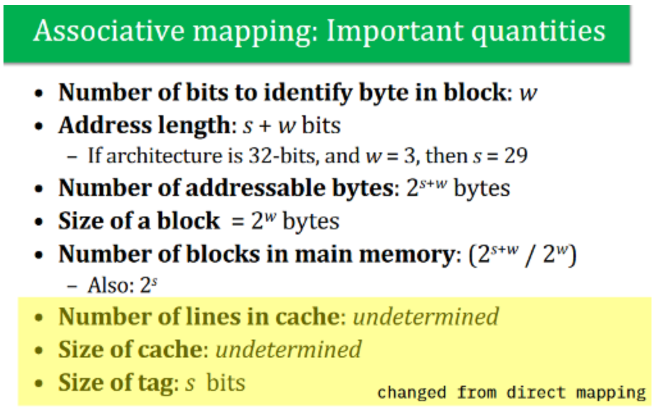
> if program references two block, one after another, which are mapped to the same cache line, the blocks will be repeatedly swapped into and out of cache
= **thrashing** (disadvantage)

Associative mapping

> each memory block may be loaded into any line of cache - the tag in cache uniquely identifies a block in memory

- To determine of a block is in the cache:
  - Simultaneously examine every line's tag for a match (comparisons in parallel)

• tag: most significant; leftmost (s - w) bits



Associative mapping: Important quantities

- **Number of bits to identify byte in block**: w
- **Address length**: $s + w$ bits
  - If architecture is 32-bits, and $w = 3$, then $s = 29$
- **Number of addressable bytes**: 2^{s+w} bytes
- **Size of a block** $= 2^w$ bytes
- **Number of blocks in main memory**: $(2^{s+w} / 2^w)$
  - Also: 2^s
- **Number of lines in cache**: *undetermined*
- **Size of cache**: *undetermined*
- **Size of tag**: s bits

changed from direct mapping

Set-Associative mapping

- we have sets of cache lines (all the same size)
> a block can be mapped into any lines of its set
> set is determined by the block #
> the associativeness of the set determines how many of the lines in the set can be used for that block

Replacement Algorithms

> what to do when cache is filled
- If direct mapping: no choice of which line
- Associative & set associative: replacement algo.

- LRU (least recently used)
- FIFO (first in first out)
- LFU (least frequently used)
- Random

Write policy

> **write through**: any write to cache line are also made to its main memory block; the into carries through
> **write back**: dirty bit is set when cache is modified - set bit signals write back to memory when line is chosen for replacement

Line size:

- when a block is loaded into the cache, other addresses around the original "cache miss" cache will also be loaded
- Under a certain line size threshold, it improves hit ratio (thus better performance), but above a line threshold, it is more likely that the extra cache loaded is not needed == hit ratio stops improving

Number of Caches:

> multi level caches are good, but L1 cache is the most influential on hit rate

Unified vs. Split caches

Unified: instructions and data share the same cache
(+) auto balances load between instruction fetch and data fetch
(-) only one cache
Split: instructions and data separate cache
(+) instruction and data fetches occur in parallel
ideal: split L1 caches, unified at other levels

Flash Memory
> form of non-volatile memory
(SRAM or DRAM)

EPROM (erasable memory)

- Default value of bits is set to 1
- Bit set to 0 via channel hot electron injection
- To set bits back to 1, must reset all by:
  - Chip exposure to ultraviolet light

EEPROM (electronically erasable memory)

- Set and reset of 1 and 0 done electronically
  - Can be done individually
- Reading, writing, erasing is very slow

NOR flash memory

- Requires much less silicon real-estate than EEPROM = higher data density

NAND flash memory
> different from NOR: data access is at page level

- Can't access individual bytes- must transfer whole page containing that byte
- Memory density is muchhh higher than NOR
- Is intended to replace external storage devices such as hard drives which already transfer data in large blocks

Wear leveling
> distribute writes/ erasure on flash memory in order that blocks wear out more or less uniformly, to prevent the same set of blocks being hit repeatedly and getting worn out more quickly (= memory errors)

Write amplification
> the actual amount of data physically written to the storage media is a multiple of the logical amount needed to be written

File Translation Layer (FTL)
> solution to above issues; is a layer of software/ hardware (algorithm) that handle:

- Mapping of logical pages to physical pages
- + other issues mentioned above, like:

> translation performance
> SRAM overhead
> block utilization
> Garbage collection
> Fault tolerance