The final exam will cover material mainly from 04-06c and on, and projects 5-8.

Table of Contents

Lecture 4: Processor Architecture - Y86-64 ISA, Logic Circuits

Instruction Set Architecture
- Assembly Language View
  - Processor state... Registers, memory...
  - Instructions
    - addq, pushq, ret....
    - How instructions are encoded as bytes
- Layers of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast.
    - E.g. execute multiple instructions simultaneously

Y86-64 Processor State
- Program Registers
  - 15 registers (omit %r15). Each 64 bits
- Condition Codes
  - Single-bit flags set by arithmetic or logical instructions
    - ZF: Zero SF: Negative OF: Overflow
- Program Counter
  - Indicates address of next instructions
- Program Status
  - Indicates either normal operation or some error condition
- Memory
  - Byte-addressable storage array
  - Words stored in little-endian byte order

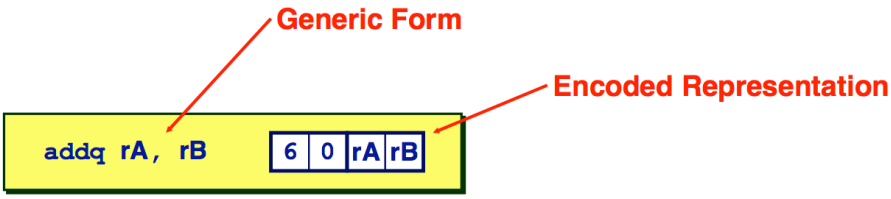
Y86-64 Instructions
- Format
  - 1 - 10 bytes of information read from memory
    - Can determine instruction length from first byte
    - Not as many instruction types, and simpler encoding than with x86-64
  - Each accesses and modifies some part(s) of the program state

Encoding Registers
- Each register has 4-bit ID
- Register ID 15 (0xF) indicates "no register"

%rax	0
%rcx	1
%rdx	2
%rbx	3
%rsp	4
%rbp	5
%rsi	6
%rdi	7

%r8	8
%r9	9
%r10	A
%r11	B
%r12	C
%r13	D
%r14	E
No Register	F

Instruction Example
- Addition Instruction



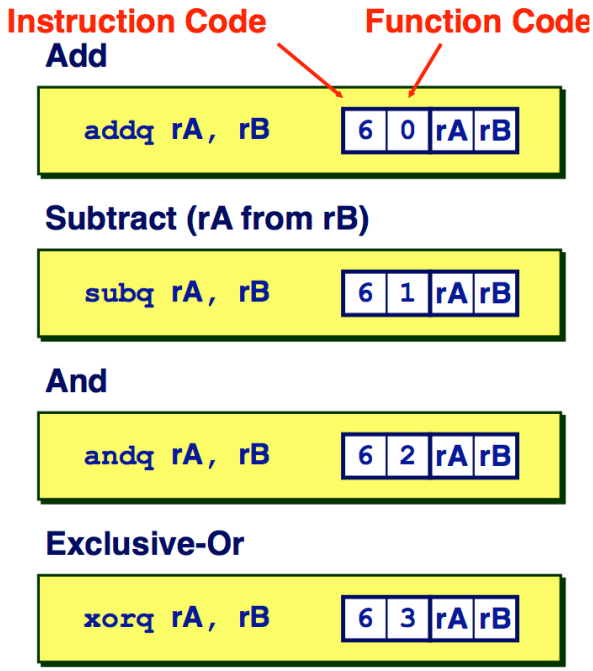
- Add value in register rA to that in register rB
  - Store result in register rB
  - Note that Y86-64 only allows addition to be applied to register data
- Set condition codes based on result
- E.g. addq, %rax, %rsi Encoding 60 06

- Two-byte encoding

- First indicates instruction type
- Second gives source and destination registers

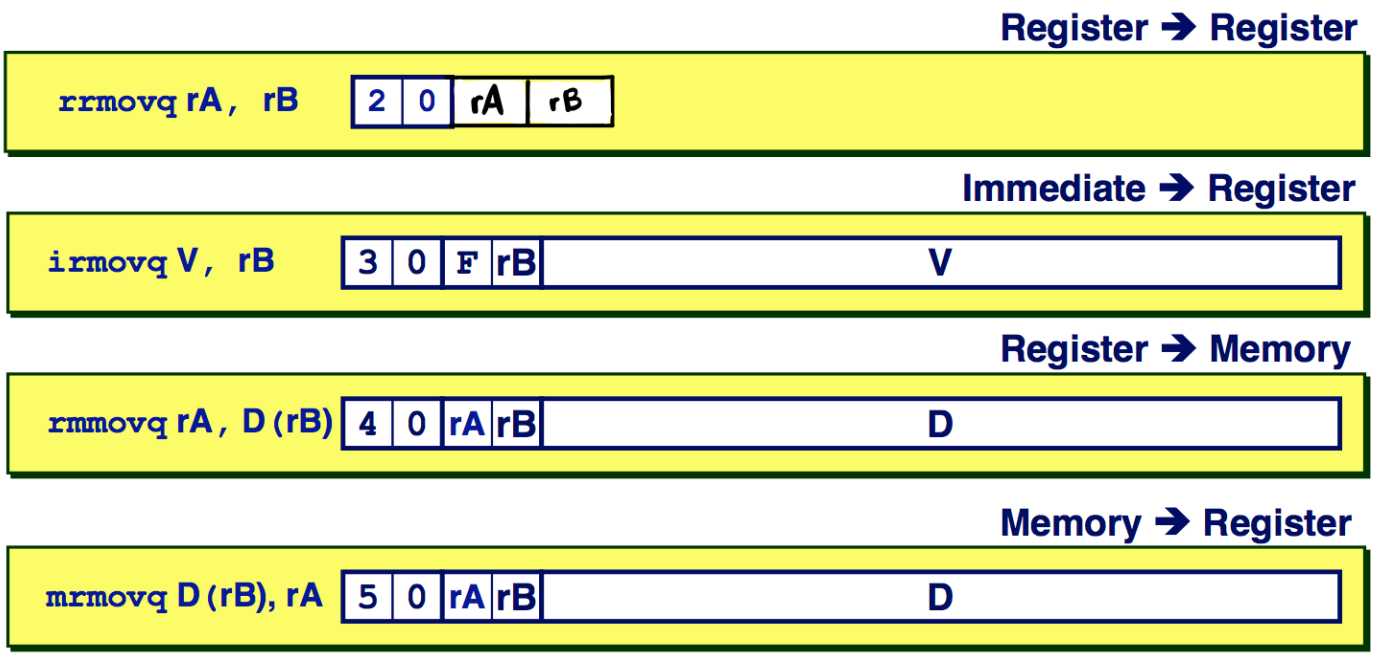
Arithmetic and Logical Operations

- Add
- Subtract
- And
- Exclusive-Or
  - Refer to generically as "OPq"
  - Set condition codes as side effects



Move Operations

- rrmovq
- irmovq
- removq
- mrmovq
  - Liike the x86-64 move instruction
  - Simpler format for memory addresses
  - Give different names to keep them distinct



Move Instruction Examples

X86-64	Y86-64
`movq $0xabcd, %rdx`	`irmovq $0xabcd, %rdx`
Encoding: `30 82 cd ab 00 00 00 00 00 00`	
`movq %rsp, %rbx`	`rrmovq %rsp, %rbx`
Encoding: `20 43`	
`movq -12(%rbp),%rcx`	`mrmovq -12(%rbp),%rcx`
Encoding: `50 15 f4 ff ff ff ff ff ff ff`	
`movq %rsi,0x41c(%rsp)`	`rmmovq %rsi,0x41c(%rsp)`
Encoding: `40 64 1c 04 00 00 00 00 00 00`	

Jump Instructions

Jump Unconditionally

jmp Dest	7	0	Dest

Jump When Less or Equal

jle Dest	7	1	Dest

Jump When Less

jl Dest	7	2	Dest

Jump When Equal

je Dest	7	3	Dest

Jump When Not Equal

jne Dest	7	4	Dest

Jump When Greater or Equal

jge Dest	7	5	Dest

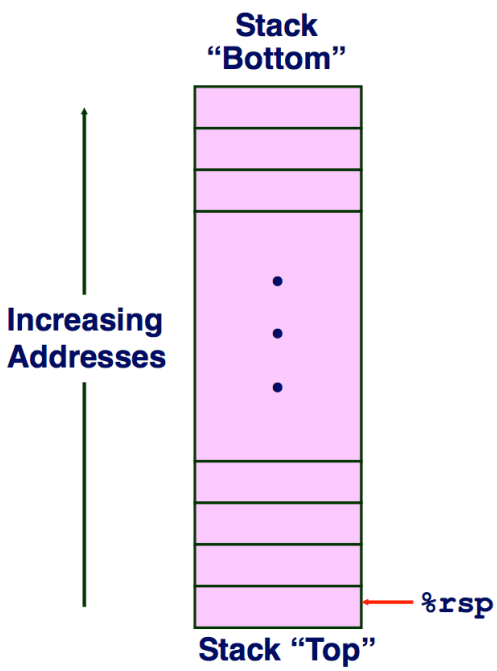
Jump When Greater

jg Dest	7	6	Dest

Y86-64 Program

Stack

- Region of memory holding data
- Used in Y86-64 supporting procedure calls
- Stack top indicated by %rsp
  - Address of top stack element
- Stack grows toward lower addresses
  - Top element is at highest address in the stack
  - When pushing, must first decrement stack pointer
  - After popping, increment stack pointer



Stack Operations

- pushq
  - Decrement %rsp by 8
  - Store word from rA to memory at %rsp
  - Like x86-64

pushq rA	A	0	rA	F

- popq
  - Read word from memory at %rsp
  - Save in rA
  - Increment %rsp by 8
  - Like x86-64

popq rA	B	0	rA	F

Subroutine Call and Return

- call
  - Push address of next instruction onto stack
  - Start executing instructions at Dest
  - Like x86-64

- ret
  - Pop value from stack
  - Use as address for next instruction
  - Like x86-64

- halt
  - Stop executing instructions
  - X86-64 has comparable instruction, but can't execute in user mode
  - We will use it to stop the simulator
  - Encoding ensures that program hitting memory initialized to zero will haunt

Y86-64 Sample Program Structure

- Program starts at address 0
- Must set up stack
  - Where located
  - Pointer values
  - Make sure don't overwrite code!
- Must initialize data

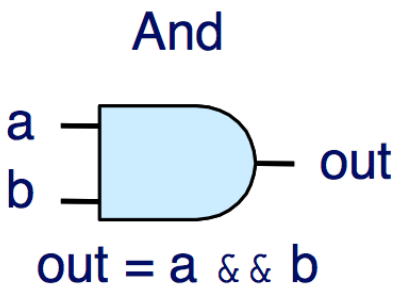
```
init:                   # Initialization
   . . .
   call Main
   halt

   .align 8             # Program data
array:
   . . .

Main:                   # Main function
   . . .
   call len   . . .

len:                    # Length function
   . . .

   .pos 0x100           # Placement of stack
Stack:
```

Overview of Logic Design

- Fundamental Hardware Requirements
  - Communication
    - How to get values from one place to another
  - Computation
  - Storage
- Bits are Our Friends
  - Everything expressed in terms of values 0 and 1
  - Communication
    - Low or high voltage on wire
  - Computation
    - Compute Boolean functions
  - Storage
    - Store bits of information

Computing with Logic Gates

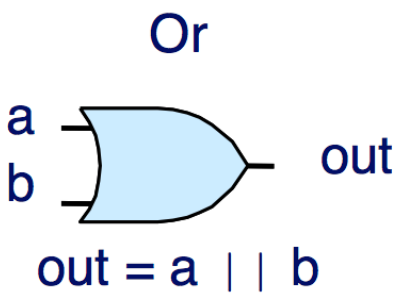
- **AND**



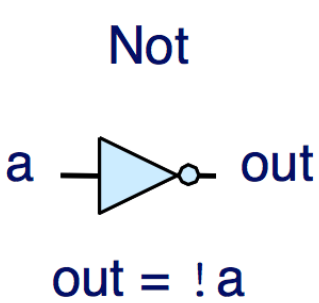
* Outputs are Boolean functions of inputs

* Respond continuously to changes in inputs

- **OR**

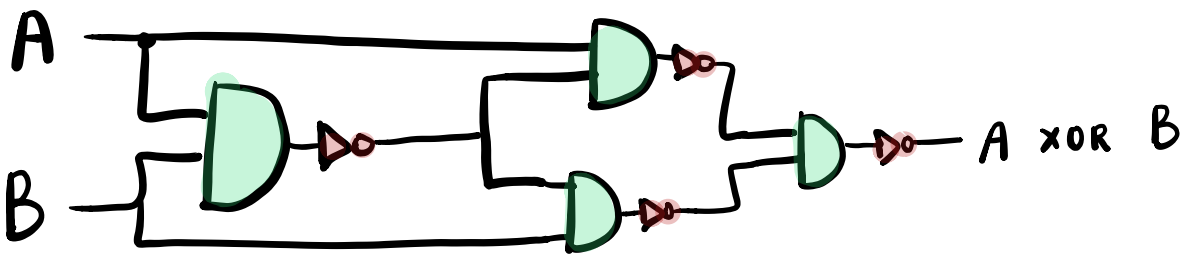


- **NOT**

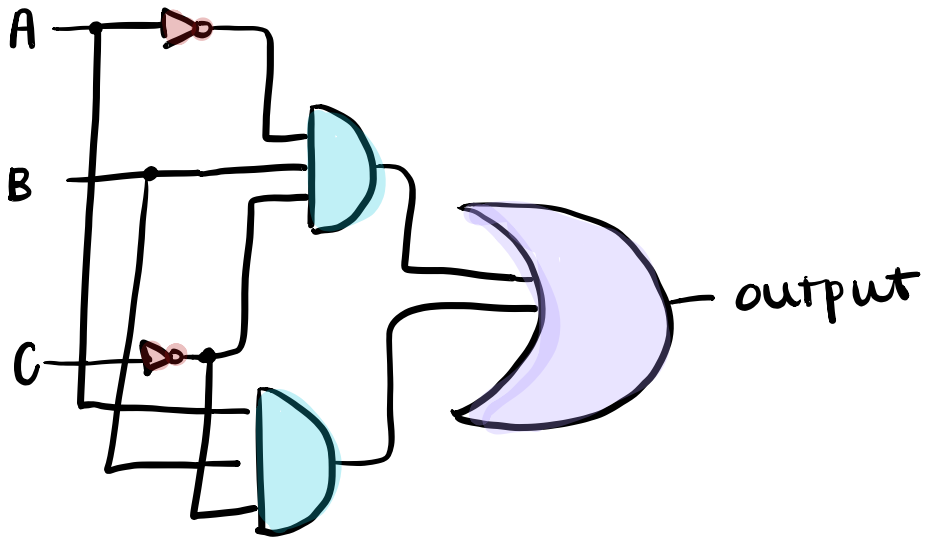


Examples:

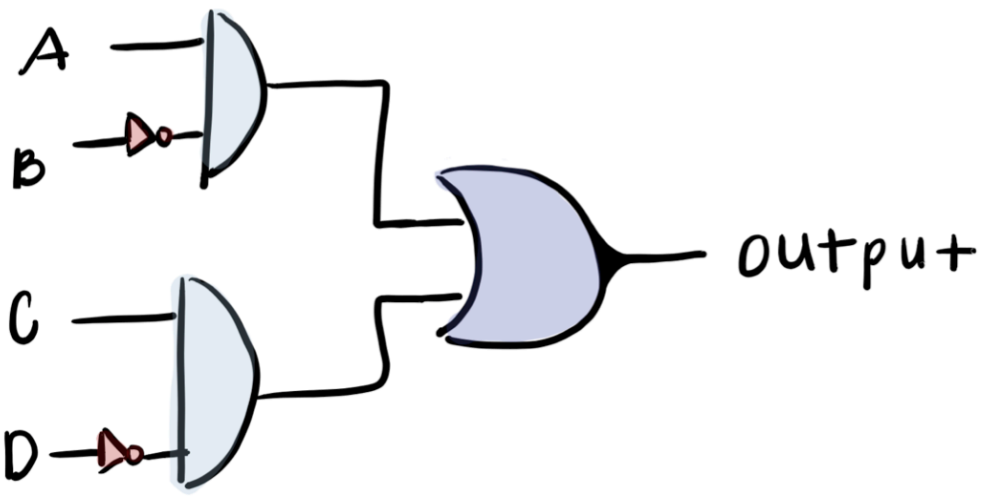
XOR



(!A && B && !C) || (A && B && !C)



(A && !B) || (C && !D)



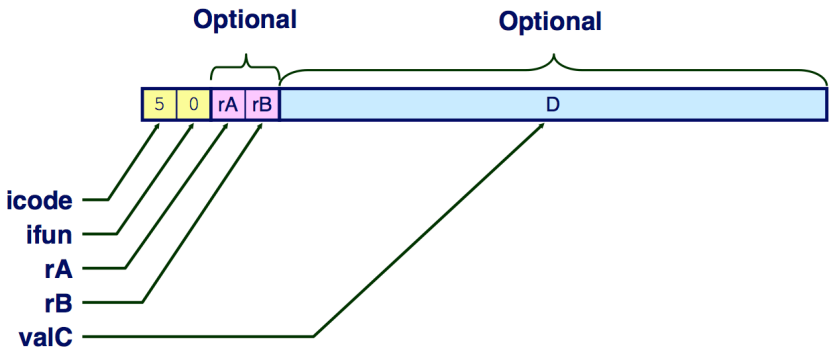
Lecture 4b: Processor Architecture - Y86-64 ISA, Logic Circuits

SEQ Stages

- Fetch
  - Read instruction from instruction memory
- Decode
  - Read program registers
- Execute
  - Compute value or address
- Memory
  - Read or write data
- Write back
  - Write program registers
- PC
  - Update program counter

Instruction Decoding

- Instruction Format
  - Instruction byte — icode : ifun
  - Optional register byte — rA : rB
  - Optional constant word — valC



Computed Values

Fetch

- **icode** Instruction code
- **ifun** Instruction function
- **rA** Instr. Register A
- **rB** Instr. Register B
- **valC** Instruction constant
- **valP** Incremented PC

Decode

- **srcA** Register ID A
- **srcB** Register ID B
- **dstE** Destination Register E
- **dstM** Destination Register M
- **valA** Register value A
- **valB** Register value B

Execute

- **valE** ALU result
- **Cnd** Branch/move flag

Memory

- **valM** Value from memory

Fetch Logic
- Predefined Blocks
  - PC: Register containing PC
  - Instruction memory: Read 10 bytes (PC to PC+9)
    - Signal invalid address
  - Split: Divide instruction byte into icode and ifun
  - Align: Get fields for rA, rB, and valC
- Control Logic
  - Need reigns: Does this instruction

Decode Logic
- Register File
  - Read ports A, B
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)
- Control Logic
  - srcA, srcB: read port addresses
  - dstE, dstM: write port addresses
- Signals
  - And: Indicate whether or not to perform conditional move
    - Computed in Execute stage

Execute Logic
- Units
  - ALU
    - Implements 4 required functions
    - Generates condition code values
  - CC
    - Register with 3 condition code bits
  - cond
    - Computes conditional
    - Jump/move flag
- Control Logic
  - Set CC: Should condition code register be loaded?
  - ALU A: Input A to ALU
  - ALU B: Input B to ALU
  - ALU fun: What function should ALU compute?

Memory Logic
- Memory
  - Reads and writes memory word
- Control Logic
  - Stat: What is instruction status?
  - Mem. Read: should word be read?
  - Mem. Write: should word be written?
  - Mem. Addr: select address
  - Mem. Data: select data

Writeback
- Register File
  - Write ports E, M
  - Addresses are register IDs or 15 (0xF) (no access)
- Control Logic
  - dstE, dstM: write port addresses
- Signals
  - Cnd: Indicate whether or not to perform conditional move
    - Computed in Execute stage

PC Update Logic

- New PC
  - Select next value of PC

Stage Computation: Arith/Log. Ops

	OPq rA, rB	
Fetch	icode:ifun ← M_1[PC] rA:rB ← M_1[PC+1] valP ← PC+2	Read instruction byte Read register byte Compute next PC
Decode	valA ← R[rA] valB ← R[rB]	Read operand A Read operand B
Execute	valE ← valB OP valA Set CC	Perform ALU operation Set condition code register
Memory		
Write back	R[rB] ← valE	Write back result
PC update	PC ← valP	Update PC

pointer to first byte of instruction

↑ means how many bytes M is.

- Formulate instruction execution as sequence of simple steps
- Use same general form for all instructions

Stage Computation: rrmovq

	`rmmovq` rA, D(rB)	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	rA:rB ← M_1[PC+1]	Read register byte
	valC ← M_8[PC+2]	Read displacement D
	valP ← PC+10	Compute next PC
Decode	valA ← R[rA]	Read operand A
	valB ← R[rB]	Read operand B
Execute	valE ← valB + valC	Compute effective address
Memory	M_8[valE] ← valA	Write value to memory
Write back		
PC update	PC ← valP	Update PC

- **Use ALU for address computation**

Stage Computation: Jumps

	jXX Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Fall through address
Decode		
Execute	Cnd ← Cond(CC,ifun)	Take branch?
Memory		
Write back		
PC update	PC ← Cnd ? valC : valP	Update PC

- **Compute both addresses**
- **Choose based on setting of condition codes and branch condition**

Stage Computation: call

	`call` Dest	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
	valC ← M_8[PC+1]	Read destination address
	valP ← PC+9	Compute return point
Decode	valB ← R[`%rsp`]	Read stack pointer
Execute	valE ← valB + –8	Decrement stack pointer
Memory	M_8[valE] ← valP	Write return value on stack
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valC	Set PC to destination

- Use ALU to decrement stack pointer
- Store incremented PC

Stage Computation: ret

	`ret`	
Fetch	icode:ifun ← M_1[PC]	Read instruction byte
Decode	valA ← R[`%rsp`]	Read operand stack pointer
	valB ← R[`%rsp`]	Read operand stack pointer
Execute	valE ← valB + 8	Increment stack pointer
Memory	valM ← M_8[valA]	Read return address
Write back	R[`%rsp`] ← valE	Update stack pointer
PC update	PC ← valM	Set PC to return address

- Use ALU to increment stack pointer
- Read return address from memory

SEQ Summary

- Implementation
  - Express every instruction as series of simple steps
  - Follow same general flow for each instruction type
  - Assembly registers, memories, predesigned combinational blocks
  - Connect with control logic
- Limitations
  - Too slow to be practical
  - In one cycle, must propagate through instruction memory, register file, ALU, and data memory
  - Would need to run clock very slowly
  - Hardware units only active for fraction of clock cycle

Real-World Pipelines: Car Washes....

Sequential

Parallel

Pipelined

Idea

- Divide process into independent stages
- Move objects through stages in sequence
- At any given times, multiple objects being processed

Assignment 5 - Y86-64

```
# Author: Alyssa Kelley
# Assignment 5
# Source: I worked on this assignment with Miguel Hernan, Anne Glickenhause, and Kristine Stecker.

# Execution begins at address 0
    .pos 0
    irmovq stack, %rsp      # This sets up the stack pointer
    call main               # This executes the main program
    halt                    # This terminates the program

# Array of 10 elements
    .align 8
array:
    .quad 0x0000000000000007
    .quad 0x0000000000000001
    .quad 0x0000000000000002
    .quad 0x0000000000000003
    .quad 0x0000000000000008
    .quad 0x000000000000000A
    .quad 0x0000000000000009
    .quad 0x0000000000000006
    .quad 0x0000000000000004
    .quad 0x0000000000000005

main:
    irmovq array, %rdi
    irmovq $10, %rsi         # $10 is the length of the array stored in rsi
    call selectionSort       # selectionSort(array)
    ret

swap:
    mrmovq    (%rdi), %rax
    mrmovq    (%rsi), %rdx
    rmmovq    %rdx, (%rdi)
    rmmovq    %rax, (%rsi)
    ret


selectionSort:

    irmovq $1, %r12
    rrmovq %rsi, %r11
    subq %r12, %r11     # This looks like the c code: len--
    irmovq $0, %rcx     # This is setting i = 0 for the c code.
    jmp L1
```

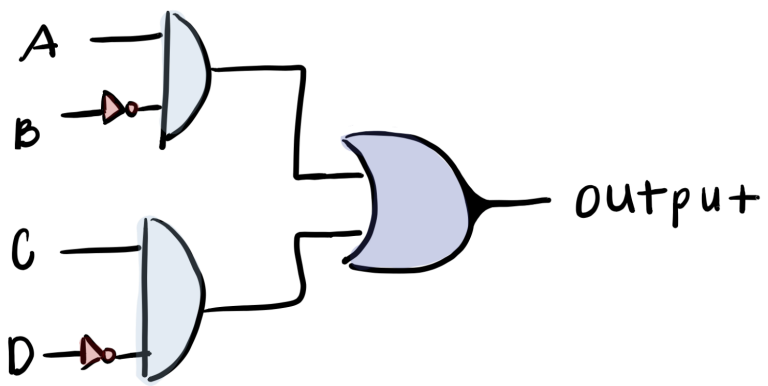
```
L2:
    # Inner loop of the c code for selectionSort.

    # C code: Get array[j]
    rrmovq %rdx, %r14
    addq   %r14, %r14
    addq   %r14, %r14
    addq   %r14, %r14       # Adding 3 times = multiplied by 8 for the sizeof(long)
    addq   %rdi, %r14       # Offset from &array

    # C code: Get array[min_idx]
    rrmovq %rbx, %r8
    addq %r8, %r8
    addq %r8, %r8
    addq %r8, %r8           # Adding 3 times = multiplied by 8
    addq %rdi, %r8          # Offset from &array

    # Dereference
    mrmovq (%r14), %r14
    mrmovq (%r8), %r8

    pushq %r14
    subq  %r8, %r14         # C code: if(array[j] < array[min_idx])
    popq  %r14
    jge L3
    rrmovq %rdx, %rbx       # C code: min_idx = j

L3:
    addq %r12, %rdx         # C code: j++
    pushq %rdx
    subq %rsi, %rdx
    popq  %rdx
    jl L2

    pushq %rcx
    subq  %rbx, %rcx        # C code: if(i==min_idx)
    popq  %rcx
    je   L4

    # This is preparing for swap
    pushq %rdi
    pushq %rsi
    rrmovq %rdi, %rsi

    # This gets array[min_idx_idx]
    rrmovq %rbx, %r13
    addq %r13, %r13
    addq %r13, %r13
    addq %r13, %r13
    addq %r13, %rdi

    # This gets array[i]
    rrmovq %rcx, %r13
    addq %r13, %r13
    addq %r13, %r13
    addq %r13, %r13
    addq %r13, %rsi

    # C code: swap(array[min_idx],array[i])
    call swap
    popq  %rsi
    popq  %rdi
```

```
111       L4:
112           addq %r12, %rcx     # C code: i++
113           pushq %rcx
114           subq %r11, %rcx     # C code: if(i < len)
115           popq %rcx
116           jl L1
117
118       L1:
119           # This is the outer loop for selectionSort
120           rrmovq %rcx, %rbx  # min_idx = i
121           rrmovq %rcx, %rdx  # j = i
122           addq  %r12, %rdx   # j++
123           jmp L2
124   ret
125
126   # Stack starts here and grows to lower addresses
127     .pos 0x220
128   stack:
```

2. Draw a circuit with inputs A, B, C and D

$(A \oplus !B) \,||\, (C \oplus !D)$

Not AND OR



3. Draw a diagram expressing how the existing how Y86-64 architecture could implement a new iaddq instruction to accomplish this functionality:

	iaddq V, rB
Fetch	icode:ifun ← M1 [PC] rA:rB ← M1 [PC+1] valC ← M8 [PC+2] valP ← PC + 10
Decode	valB ← R [rB]
Execute	valE ← valC + valB set CC
memory	
Write back	R [rB] ← valE
PC update	PC ← valP

Lecture 5: Optimizing Program Performance - Limitation of Optimizing Compilers

Performance Realities

- There is more to performance than asymptotic complexity
- Constant factors matter too
  - Easily see 10 : 1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - Algorithm, data representations, procedures and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying case modularity and generality

Optimizing Compilers

- Provide efficient mapping of program to machine
  - Register allocation
  - Code selection and ordering (scheduling)
  - Dead code elimination
  - Eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - Up to programmer to select best overall algorithm
  - Big-O savings are (often) more important than constant factors
    - But constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - Potential memory aliasing
  - Potential procedure side-effects

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer version of GCC do inter procedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

Generally Useful Optimizations

- Optimizations that you or the compiler should do regardless of processor / compiler
- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

```
    long j;
    long ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

Reduction in Strength
- Replace costly operation with simpler one
- Shift, add instead of multiply or divide
  16 * x ---> x << 4
  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```



```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

Share Common Subexpressions
- Reuse portions of expressions
- GCC will do this with -O1

Optimization Blocker #1: Procedure Calls
- Procedure to Convert String to Lower Case

```
void lower(char *s)
{
  size_t i;
  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Lower Case Conversion Performance
- Time quadruples when double string length
- Quadratic performance

Calling Strlen
- Strlen performance
  - Only way to determine length of string is to scan its entire length, looking for null character
- Overall performance, string of length N
  - N calls to strlen
  - Require times N, N-1, N-2, ... 1
  - Overall O(N^2) performance

```
/* My version of strlen */
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++;
       length++;
    }
    return length;
}
```

Improving Performance
- Move call to strlen outside of loop
- Since result does not change from one iteration to another
- Form of code motion

```
void lower(char *s)
{
  size_t i;
  size_t len = strlen(s);
  for (i = 0; i < len; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
}
```

Optimization Blocker: Procedure Calls

- Why couldn't compiler move strlen out of inner loop?
  - Procedure may have side effects
    - Alters global state each time called
  - Function may not return same value for given arguments
    - Depends on other parts of global state
    - Procedure lower could interact with strlen
- Warning:
  - Compiler treats procedure call as a black box
  - Weak optimizations near them
- Remedies
  - Use of inline functions
    - GCC does this with -O1
      - With single file
  - Do you own code motion

```
size_t lencnt = 0;
size_t strlen(const char *s)
{
    size_t length = 0;
    while (*s != '\0') {
       s++; length++;
    }
    lencnt += length;
    return length;
}
```

Lecture 5b: Optimizing Program Performance - Modern Processors, Loop Unrolling, Branch Prediction

Exploiting Instruction-Level Parallelism

- Need general understanding of modern processor design
  - Hardware can execute multiple instructions in parallel
- Performance limited by data dependencies
- Simple transformations can yield dramatic performance improvement
  - Compilers often cannot make these transformations
  - Lack of associativity and distributivity in floating-point arithmetic

Basic Optimizations

```
void combine4(vec_ptr v, data_t *dest)
{
  long i;
  long length = vec_length(v);
  data_t *d = get_vec_start(v);
  data_t t = IDENT;
  for (i = 0; i < length; i++)
    t = t OP d[i];
  *dest = t;
}
```

- Move vec_length out of loop
- Avoid bounds check on each cycle
- Accumulate in temporary

- Eliminated sources of overhead in loop

Loop Unrolling (2 x 1)

```
void unroll2a_combine(vec_ptr v, data_t *dest)
{
    long length = vec_length(v);
    long limit = length-1;
    data_t *d = get_vec_start(v);
    data_t x = IDENT;
    long i;
    /* Combine 2 elements at a time */
    for (i = 0; i < limit; i+=2) {
        x = (x OP d[i]) OP d[i+1];
    }
    /* Finish any remaining elements */
    for (; i < length; i++) {
        x = x OP d[i];
    }
    *dest = x;
}
```

- **Perform 2x more useful work per iteration**

Unrolling & Accumulating

- Idea
  - Can unroll to any degree L
  - Can accumulate K results in parallel
  - L must be multiple of K
- Limitations
  - Diminishing returns
    - Cannot go beyond throughout limitations of execution units
  - overhead for short lengths
    - Finish off iterations sequentially

Assignment 6 - Optimization

```c
// Author: Alyssa Kelley
// Assignment 6 - Part 1
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 800

typedef int array_t[N][N];

int dim()
{
    return N;
}
void old_f(array_t a, int x, int y)
{
    for (int i = 0; i < dim(); ++i)
    {
        for (int j = 0; j < dim(); ++j)
            {
                a[i][j] = i * x * y + j; // Change this part!
            }
    }
}
/*
Task: Rewrite the above procedure f to minimize unnecessary function calls and multiplications. Note that you are not required to replace the a[i][j] lookups with pointer arithmetic; focus on
the i * x * y + j equation with respect to removing unnecessary multiplications. Also write a main() function to test your procedure. Name your source file 6-1.c.
*/


void _f(array_t a, int x, int y)
{
    int *beginning = &a[0][0]; // starting the array at the beginning location
    int len = dim(); /* Creating the length as a variable so it does not have to waste that time
                        continuing to reference another function (dim()) */
    int mul_xy = x*y; // Pulling this multiplication out of loop compisition to further optimize
    int mul_xyi;
    int i = 0; // initalizing here so it is less costly in the for loops
    int j = 0;

    for (i = 0; i < len; i++)
    {
        mul_xyi = mul_xy * i; // Pulling the multiplication outside of the inner for loop.
        for (j = 0; j < len; j++)
        {
            //*pointer = mul_xyi + j; // a[i][j] = i*x*y + j
            a[i][j] = mul_xyi + j; // Only adding here because it is less costly.
        }
    }
}

```

```
51 void print_array(array_t a) // This is a function just to print the array in an ideal format.
52 {
53     printf("{ ");
54     for (int i = 0; i < N; ++i)
55     {
56         printf("{");
57 
58         for (int j = 0; j < N; ++j)
59         {
60             if (j < N-1)  // Only printing a comma after the number if it is not the last one in the sub arrays.
61             {
62                 printf("%i, ", a[i][j]);
63             }
64             else
65             {
66                 printf("%i ", a[i][j]);
67             }
68         }
69 
70         if (i < N-1)
71         {
72             printf("}, ");  // Only printing a comma after each inner array if it is not the last array within the
                   big array.
73         }
74         else
75         {
76             printf("} ");
77         }
78     }
79 
80     printf("}\n");
81 }
82 
83 int main()
84 {
85     clock_t start, end;
86     double total_time;
87     array_t array_1;
88     array_t array_2;
89 
90     int x = 2;
91     int y = 2;
92 
93     start = clock();
94     for(int i=0; i<1000; i++)
95     {
96         old_f(array_1, x, y);
97     }
98     //old_f(a, x, y);
99     end = clock();
100    total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
101    printf("This is the unoptimized run time for array a: %f\n", total_time); // you can divde here by 1000 to get
           an average
102    //print_array(a);
103 
104    // Optimized time for array a
105    start = clock();
106    //_f(a, x, y);
107    for(int i=0; i<1000; i++)
108    {
109        _f(array_1, x, y);
110    }
111    end = clock();
112    //print_array(a);
113    total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
114    printf("This is the optimized run time for array a: %f\n", total_time);
115    //print_array(a);
116
```

```
    start = clock();
    // old_f(b, x, y);
    for(int i=0; i<1000; i++)
    {
        old_f(array_2, x, y);
    }
    end = clock();
    total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    //print_array(b);
    printf("This is the unoptimized run time for array b: %f\n", total_time);

    start = clock();
    //_f(b, x, y);
    for(int i=0; i<1000; i++)
    {
        _f(array_2, x, y);
    }
    end = clock();
    total_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    //print_array(b);
    printf("This is the optimaized run time: %f\n", total_time);

    return 0;
}

```

```c
// Author: Alyssa Kelley
// Assignment 6 - Part 2c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 20000 // Length of array

/*
# u in %rbx, v in %rax, length in %rcx, i in %rdx, sum in %xmm1
.L87:
movss (%rbx, %rdx, 4), %xmm0 # Get u[i]
mulss (%rax, %rdx, 4), %xmm0
adds %xmm0, %xmm1
addq $1, %rdx
cmpq %rcx, %rdx
jl .L87
# Multiply by v[i] #Addtosum
# Increment i
# Compare i to length # If <, keep looping)
*/

void inner(float *u, float *v, int length, float *dest)
{ // This is the unoptimized function.
    int i;
    float sum = 0.0f;
    for (i = 0; i < length; ++i)
        {
            sum += u[i] * v[i];
        }
    *dest = sum;
}



void inner2(float *u, float *v, int length, float *dest)
{
    // four-way loop unrolling with four parallel accumulators (which is the contents of the for loop)
    int i;              // Intialize as much as possible before the for loop to get the most optimization.
    float sum1 = 0.0f;  // Setting them all to start at 0.
    float sum2 = 0.0f;
    float sum3 = 0.0f;
    float sum4 = 0.0f;
    int limit = length - 3; // Offsetting this to subtract 3 to reflect the 4 we are looking at at one time.

    for(i = 0; i < limit; i += 4) // pull out 4 at a time as just mentioned above.
    {
        // implementing four-way loop unrolling. See Lecture 5b Slide # 14/18
        sum1 = sum1 + u[i] * v[i];     // Creating a sum for each 4 we pull out.
        sum2 = sum2 + u[i+1] * v[i+1];
        sum3 = sum3 + u[i+2] * v[i+2];
        sum4 = sum4 + u[i+3] * v[i+3];
    }
    // This is cleaning up the array, Source: Lecture 5b, Slide 18.
    for(; i < length; i++)
    {
        sum1 = sum1 + (u[i] * v[i]);
    }
    *dest = sum1 + sum2 + sum3 + sum4; // destination is used to create a new array.
}

```

```c
int main(int argc, char *argv[])
{
    clock_t start1; /* Initalizing the start and end times for both the inner
    function (start1 / end1) and the optimized inner2 funciton (start2 / end2) */
    clock_t start2;
    clock_t end1;
    clock_t end2;
    float *u = malloc(sizeof(float) * N); // Mallocing
    float *v = malloc(sizeof(float) * N);
    float dest = 0;
    int num_func_calls = 1000; // This is the amount of how many times we run the functions.

    for (int i=0; i<N; i++)
    {
        u[i]=i;  // This is filling the arrays as large as N which is a
        v[i]=i;  // global variable defined at the top.
    }

    start1 = clock();
    for(int i=0; i<num_func_calls; i++) /* This is calling the function 1000 times (num_func_calls) so we can have a large, and
                                more accurate amount of data. You can then divide the total_time by 1000 to get
                                the average time it takes to run it once. */

    {
        inner(u, v, N, &dest);
    }
    end1 = clock();


    double total_time1 = ((double)(end1 - start1)) / CLOCKS_PER_SEC; // This is calculating the time it took to execute the function.
    printf("This is the unoptimized run time for array a: %f\n", total_time1/num_func_calls); // Average time for one funciton call
    printf("This is the sum for inner: %f\n", dest);

    start2 = clock();
    for(int i=0; i<num_func_calls; i++)
    {
        inner2(u, v, N, &dest); // Optimized function.
    }
    end2 = clock();
    double total_time2 = ((double)(end2 - start2)) / CLOCKS_PER_SEC;
    printf("This is the unoptimized run time for array a: %f\n", total_time2/num_func_calls);
    printf("This is the sum for inner2: %f\n", dest);

    printf("This means this was %f times faster!\n", (total_time1-total_time2)*1000);

    return 0;
}
```

The x86-64 assembly code for the inner loop is as follows:

```
# u in %rbx, v in %rax, length in %rcx, i in %rdx, sum in %xmm1
.L87:
    movss (%rbx, %rdx, 4), %xmm0 # Get u[i]
    mulss (%rax, %rdx, 4), %xmm0 # Multiply by v[i]
    adds %xmm0, %xmm1            # Add to sum
    addq $1, %rdx                # Increment i
    cmpq %rcx, %rdx              # Compare i to length
    jl .L87                      # If <, keep looping
```

load

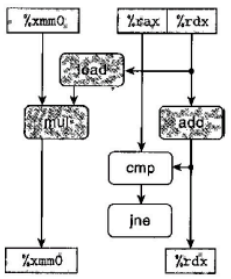
load

Registers to include:

rax, rbx
rdx, rcx
xmm0, xmm1

From the textbook:

Figure 5.14
Abstracting combine4 operations as a data-flow graph. We rearrange the operators of Figure 5.13 to more clearly show the data dependencies (a), and then further show only those operations that use values from one iteration to produce new values for the next (b).

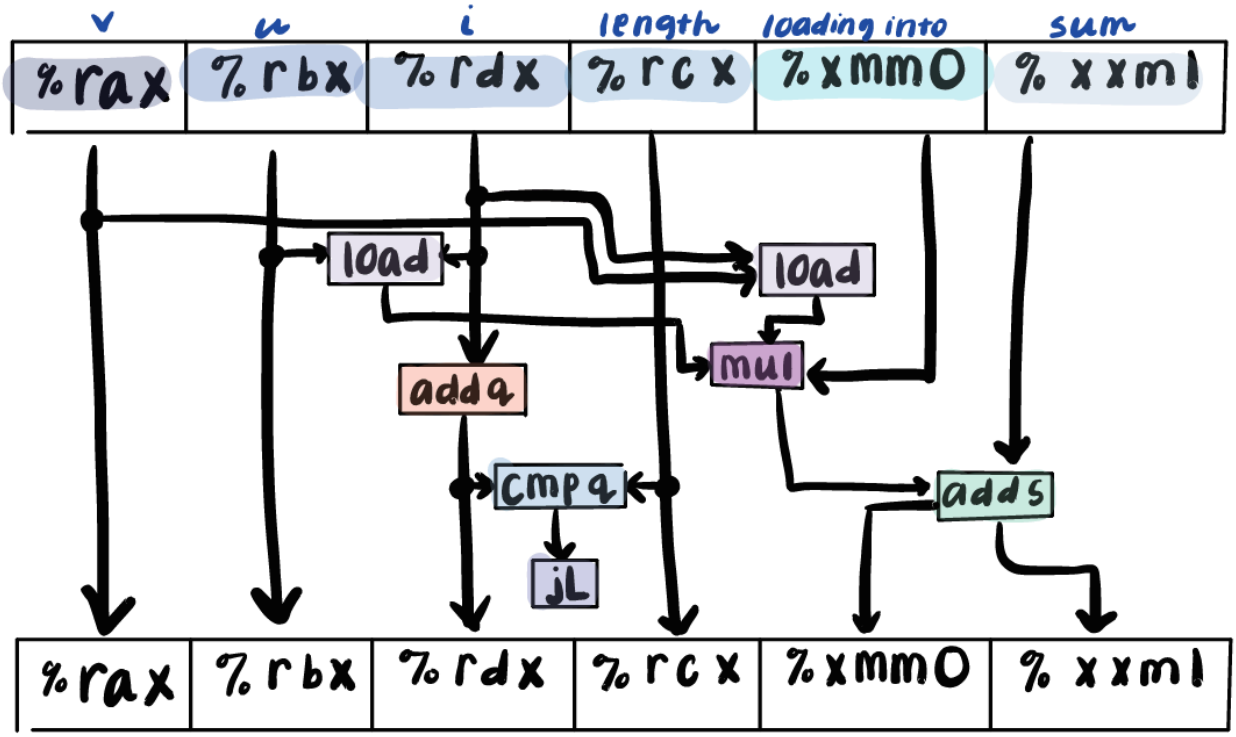


(a)

following this example.

Instructions:

load
addq, adds
cmpq



1) 2)

b. (20) Which operation(s) in the loop can NOT be pipelined? Why? What are the latencies of these operations? 3) Based on this, what is the lower latency bound (in terms of CPE) of the procedure? Assume that *float* addition has a latency of 3, *float* multiplication has a latency of 5, and all integer operations have a latency of 1. Hint: think about which operation(s) depend on a result from the previous loop iteration. Write your answers in your solutions document.

1) The operations in the loop that cannot be pipelined are those that rely on the previous / returned values. An example of this in the scenario above would be the incrementer i → %rdx, and the two add instructions use i to iterate through the loop and compare require i so these all can not be pipelined.

to reiterate, the add instructions are what cannot be pipelined.

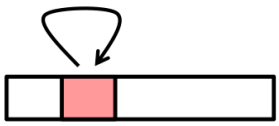
2) The two latencies for these operations are 1 for the integer latency (adds) and 3 for the float addition latency (addq).

3) The lower bound latency is 3.

Lecture 6: The Memory Hierarchy - Memory Technologies, Locality

Random-Access Memory (RAM)

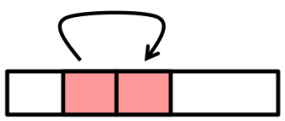
- Key feature



  - RAM is traditionally packages as a chip
  - Basic storage unit is normally a cell (one bit per cell)
  - Multiple RAM chips from a memory
- RAM comes in two varieties
  - SRAM (Static RAM)
  - DRAM (Dynamic RAM)

Locality

- Principle of Locality: Programs tend to use data and instructions with addresses near or equal to those they have used recently



- **Temporal locality:**
  - Recently referenced items are likely to be referenced again in the near future
- **Spatial locality:**
  - Items with nearby addresses tend to be referenced close together in time

Locality Example

- Data reference
  - Reference array elements in succession — **Spatial locality**
  - Reference variable sum each iteration — **Temporal locality**
- Instruction references
  - Reference instructions in sequence — **Spatial locality**
  - Cycle through loop repeatedly — **Temporal locality**

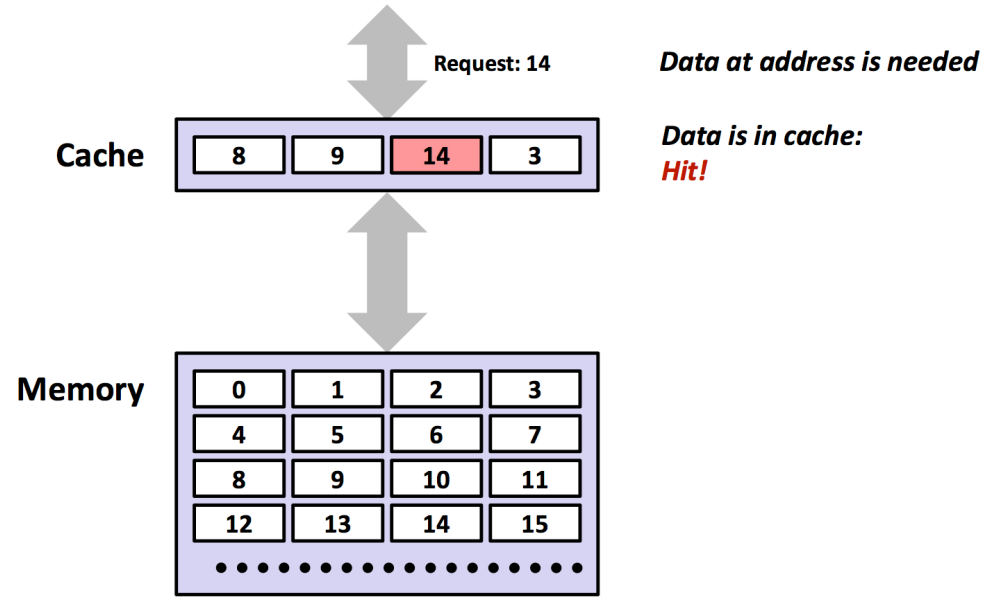
Qualitative Estimates of Locality

- Claim: Being able to look at code and get a qualitative sense of its locality is a key skill for a professional programmer

Lecture 6b: The Memory Hierarchy - Caching in the Memory Hierarchy

Memory Hierarchies

- Some fundamental and enduring properties of hardware and software
  - The gap between CPU and main memory speed is widening
  - Fast storage technologies cost more per byte, have less capacity, and require more power (heat!)
  - Well-written programs tend to exhibit good locality
- These fundamental properties complement each other beautifully
- They suggest an approach for organizing memory and storage systems knowns as a **memory hierarchy**

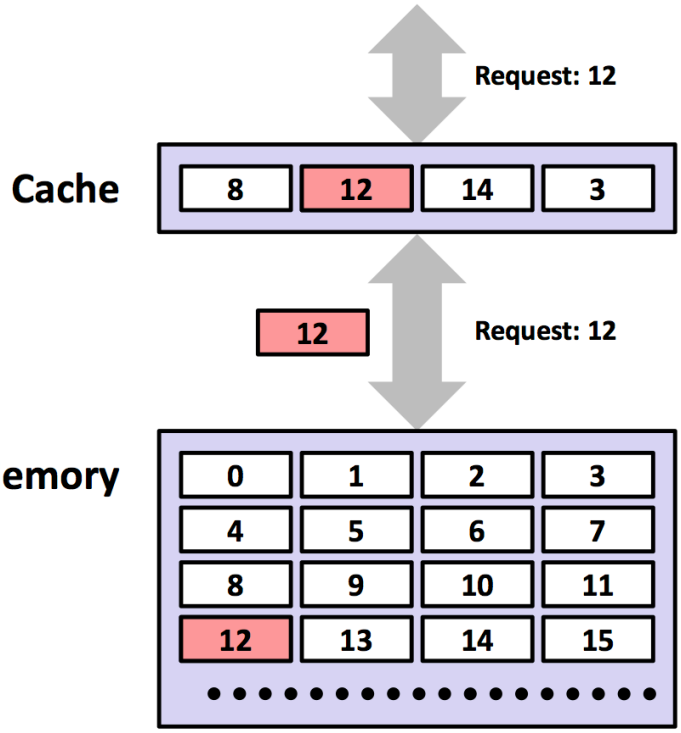


Cache

- Cache: A smaller, faster storage device that acts as a staging area for a subset of the data in a larger, slower device
- Fundamental idea of a memory hierarchy:
  - For each k, the faster, smaller device at level k serves as a cache for the larger, slower device at level k + 1
- Why do memory hierarchies work?
  - Because of locality, programs tend to access the data at level k more often than they access the data at level k + 1
  - Thus, the storage at level k + 1 can be slower, and thus larger and cheaper per bit
- Big idea: The memory hierarchy creates a large poop of storage that costs as much as the cheap storage near the bottom, but that serves data to programs at the rate of the fast storage near the top

Hit!

General Cache Concepts: Miss



Data at address is needed

Data is not in cache:
Miss!

Data is fetched from memory

Data is stored in cache
- Replacement policy: determines which block gets evicted (victim)

Miss!

	Tag	Set	Offset BLOCK	
1-way	3	2	3	direct map
2-way	4	1	3	2 way associative cache

	valid	tag
set 0	~~0~~ 1	~~?~~ 0000
	~~0~~ 1	~~?~~ 0010
set 1	~~0~~ 1	~~?~~ 0000
	0	?

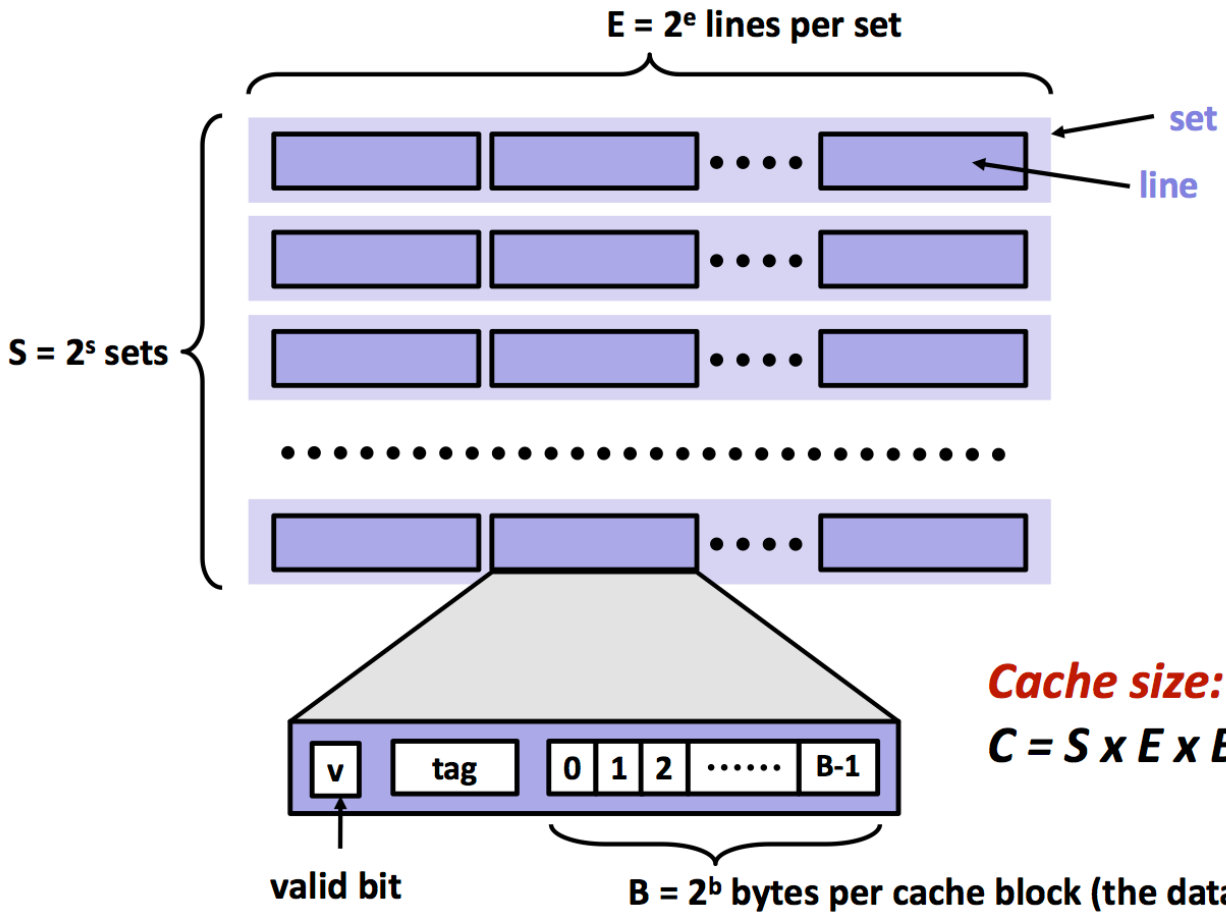
addresses

0x02	0000	0	010	miss (always miss b/c nothing there)
0x04	0000	0	100	hit (don't load b/c it is already in there)
0x08	0000	1	000	miss
0x26	0010	0	110	miss
0x04	0000	0	100	hit
0x24	0010	0	100	hit

Example:

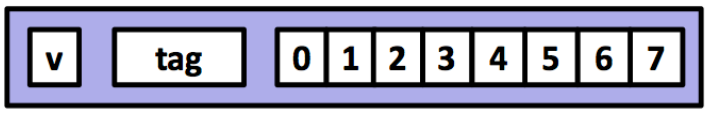
tag set offset

General Cache Organization (S, E, B)

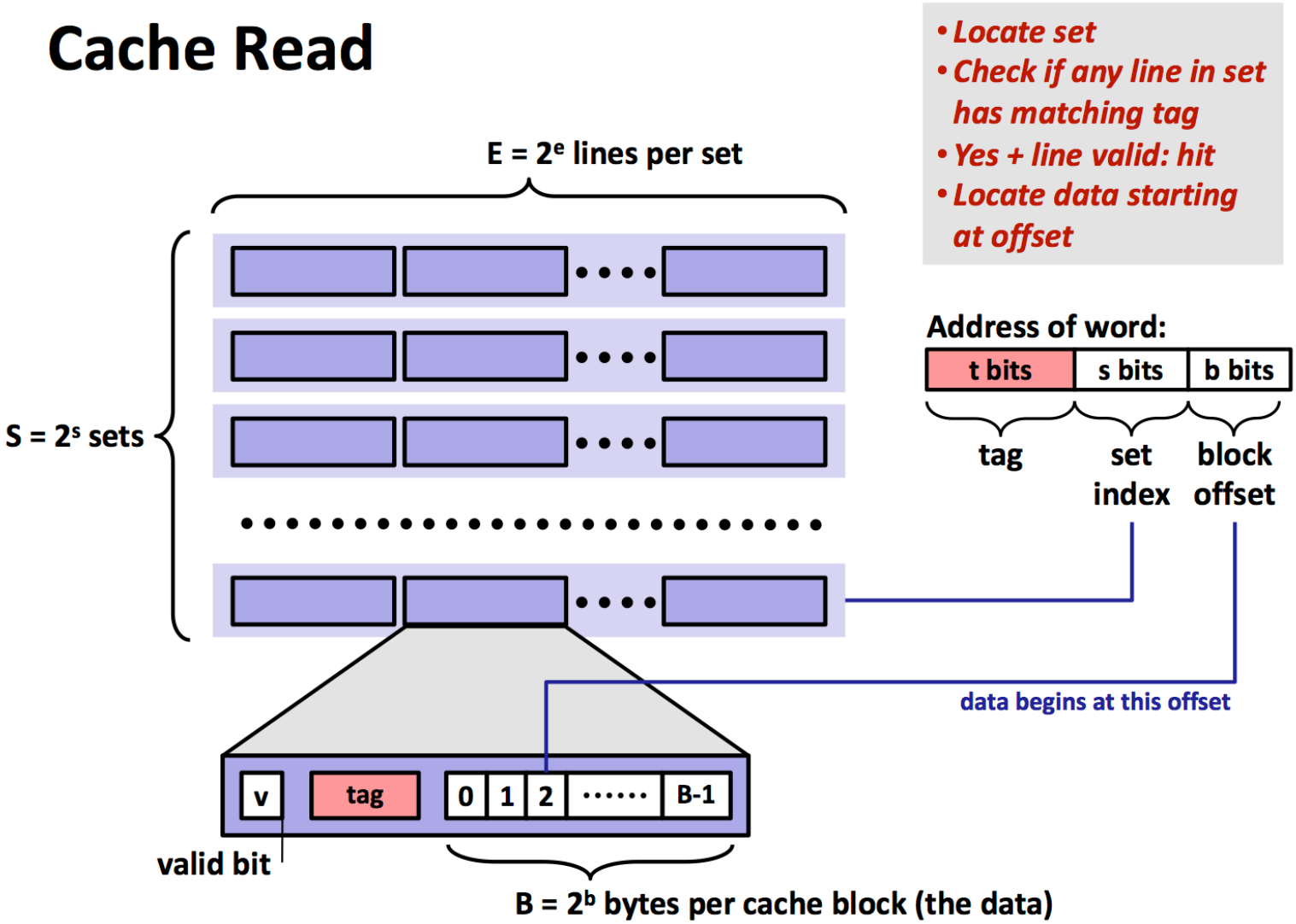


Cache size:

C = S x E x B data bytes



Decoding Addresses



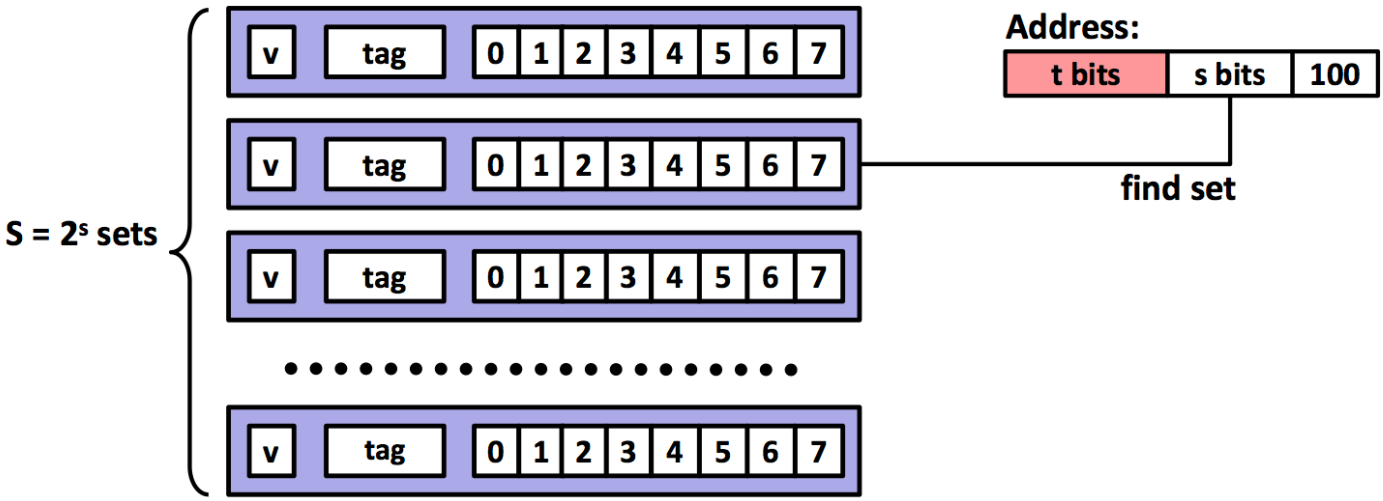
- Every byte in memory belongs to a block:
  - Lowest order o "offset" bits specify the byte's position in its block
- Each block belongs to a set in the cache:
  - Next lowest order s "set" bits specify the cache set to which the block belongs
- Many many blocks in memory will belong to the same cache set, so we need a way to identify cached blocks:
  - Remaining t "tag" bits are used to determine if a cached block is the block containing the desired byte

Cached Blocks

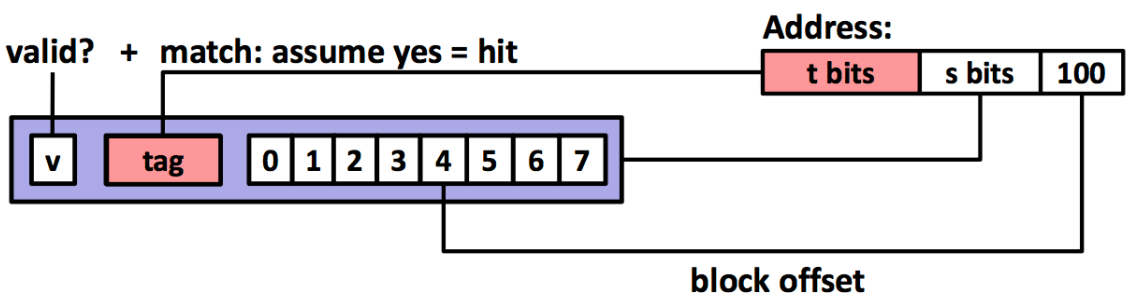
- Each cached block is represented in the cache by:
  - The bytes in the block
  - The block's identifying tag (the high order bits of all addresses in the block)
  - A valid bit (can't determine if the tag and block bytes were written intentionally by looking at the tag and block bytes alone)

Example: Direct Mapped Cache (E = 1)

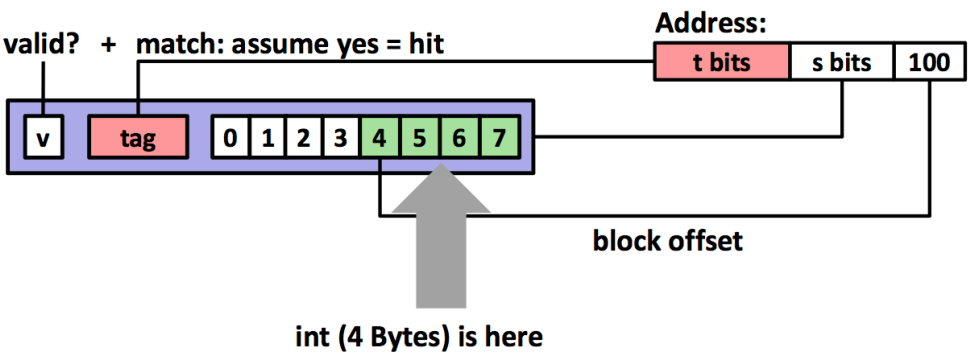
Direct mapped: One block per set
Assume: cache block size 8 bytes



Direct mapped: One block per set
Assume: cache block size 8 bytes



Direct mapped: One block per set
Assume: cache block size 8 bytes



---> Basically, you have your sets, and in those sets, you will be able to store the set and the corresponding tag. Then you will continue to look at the next addresses and if that saved set has the same tag, then it is a hit! If not, then it is a miss, and that missed information is stored in the cache set if there is room. If there is no room, it kicks out the oldest saved set and tag and replaces it with this recently missed on.

If no match: old line is evicted and replaced

Direct-Mapped Cache Simulation

t=1	s=2	b=1
x	xx	x

This will be on the final!!

M=16 bytes, 4-bit addresses,
8-byte cache, B=2 bytes/block,
E=1 blocks/set, S=4 sets

Address trace (reads, one byte per read):

0	**[0$\underline{00}$0$_2$],**	miss
1	**[0$\underline{00}$1$_2$],**	hit
7	**[0$\underline{11}$1$_2$],**	miss
8	**[1$\underline{00}$0$_2$],**	miss
0	**[0$\underline{00}$0$_2$]**	miss

	v	Tag	Block
Set 0	1	0	M[0-1]
Set 1	0	?	?
Set 2	0	?	?
Set 3	1	0	M[6-7]

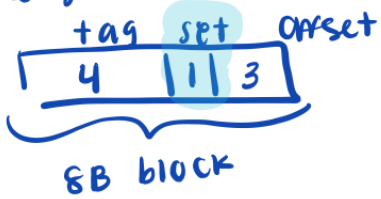
2. [25] Consider a 16B direct-mapped cache with 8B blocks and 2 sets for an 8-bit architecture (i.e., 256 bytes of memory):

c. (15) Consider the following sequence of memory accesses. For each address, show the tag, set, offset, and whether it resulted in hit or miss:

		tag	set	offset	hit / miss	
0000 0000	0x00	0000	0	000	miss	
0000 0100	0x04	0000	0	100	hit	
0000 1000	0x08	0000	1	000	miss	(nothing in set 1)
0000 0100	0x04	0000	0	100	hit	
0000 0000	0x00	0000	0	000	hit	

→ explaination for why hit / miss.

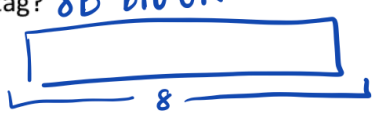
cache set = 1

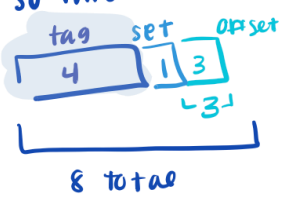


b. (5) How many address bits are used to represent the cache tag? 8B block

2^3 offset = 8

So this would be 3 bits for the offset





cache tag = 4

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// Author: Alyssa Kelley
// Assigment 7 - Part 2

/* Notes for 7-2:

0 x 1  2  3  4  5  6   7    8
  |_________________|  |_|  |_|
          tag          set offset
*/

unsigned int getOffset(unsigned int address)
{
  /* This is going to return the right most bit.
     Example: 0x12345678 -> Offset = 8
  */
    unsigned int bitmask = 0xF; // Using the mask 1111 to get the last 4 bits to indicate the last number.
  unsigned int offset = address & bitmask; // & these together will have 0 everyhting else except for the last 4 bits.
    return offset; // Returning the right most 4 bits.
}

unsigned int getTag(unsigned int address)
{
  /* This is going to return the left most 4*6 bits.
     Example: 0x12345678 -> Tag = 123456
  */
    unsigned int bitmask = 0xFFFFFF00; /* Using the bit mask 1111 1111 1111 1111 1111 1111 0000 0000
  and the reason for this is to get only the first 6 integers and then the last two should be dropped. */
  unsigned int set = (address & bitmask);
    return set >> 8; // Shifting off 8 so get rid of the last 2 0'd out integers.
}

unsigned int getSet(unsigned int address)
{
  unsigned int bitmask = 0x000000F0; /* Using the mask 0000 0000 0000 0000 0000 0000 1111 0000 and
  this will get the second to last integer in the hex number. */
  unsigned int tag = address & bitmask;
  return tag >> 4; // This shifts off the last integer before the second to last
}


int main()
{
  unsigned int address1 = 0x12345678;
  unsigned int address2 = 0x87654321;

  printf("0X%.8X: offest: %d - tag: %X - set: %X\n", address1, getOffset(address1), getTag(address1), getSet(address1));
  printf("0X%.8X: offest: %d - tag: %X - set: %X\n", address2, getOffset(address2), getTag(address2), getSet(address2));

    return 0;
}
```

Another cache example:

A. (15) What is the total number of memory reads? Why? It may help to think in terms of movq instructions.

Loop iteration - (for loop #1 = 16) * (for loop #2 = 16)

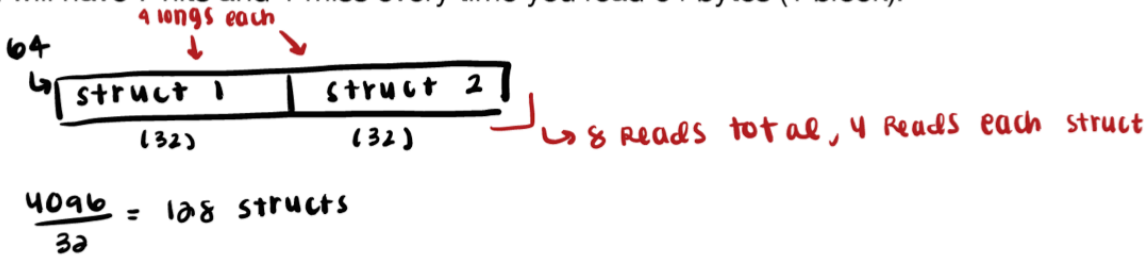
4 memory reads so (4) * (16 * 16) = 1024 total reads

Answer: 1024 total reads.

each instruction = 8 bytes and how many Reads is how many times an instruction

B. (15) What is the total number of memory reads that miss in the cache? Why?

You will have 7 hits and 1 miss every time you read 64 bytes (1 block).

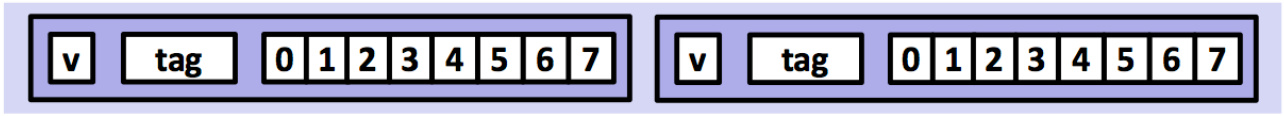


sizeof(colorPoint) = 32 bytes
Each block can hold 2 structs (because it is 64)

Explanation: Since we are reading 1024 total reads and every time we read 64 bytes then we have 1 miss so that would be a total of 16 misses and 112 hits per 1024 reads.

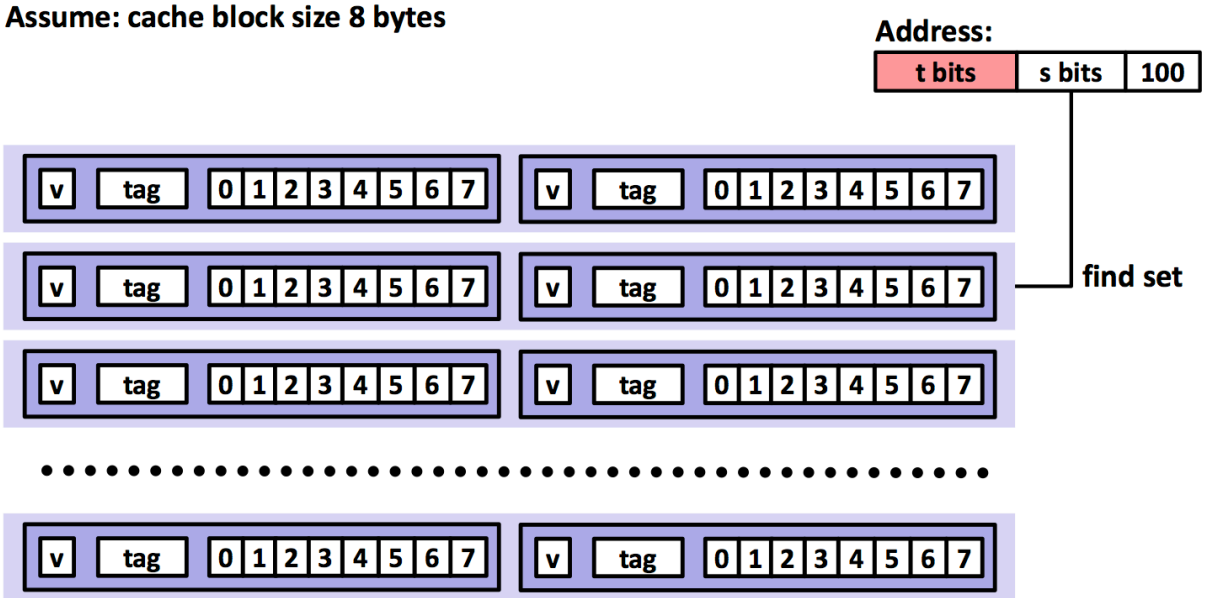
C. (10) What is the miss rate?

-> 16/128 = 1/8 = 12.5% miss rate for 1024 total memory reads.



E-way Set Associative Cache (Here: E = 2)

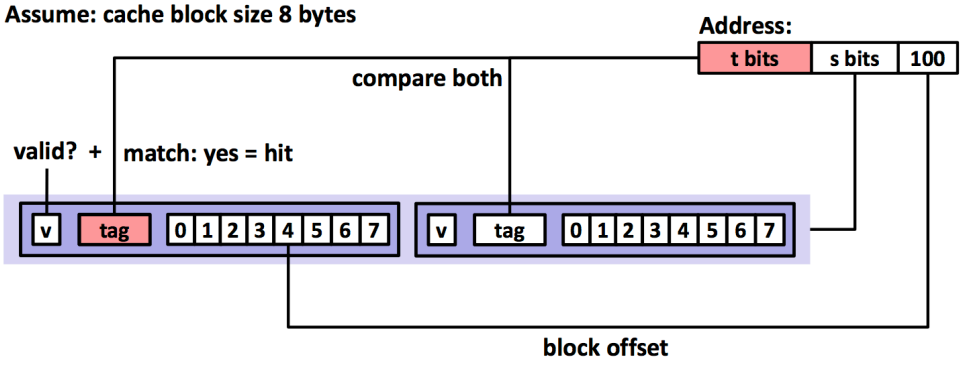
E = 2: Two blocks per set
Assume: cache block size 8 bytes



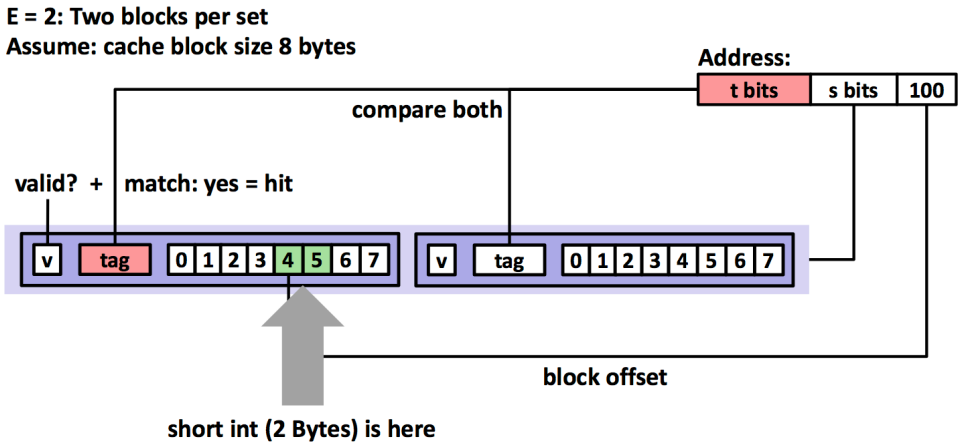
Assignment 7 - Bitwise for Tag / Set / Offset

E-way Set Associative Cache (Here: E = 2)

E = 2: Two blocks per set
Assume: cache block size 8 bytes



E-way Set Associative Cache (Here: E = 2)



Lecture 6c: The Memory Hierarchy - Direct Mapped Caches, Set-Associative Caches

Improving Temporal Locality

If no match:

- **One block in set is selected for eviction and replacement**
- **Replacement policies: random, least recently used (LRU), …**

2-Way Set Associative Cache Simulation

t=2	s=1	b=1
xx	x	x

M=16 bytes, 4-bit addresses,
8-byte cache, B=2 bytes/block,
E=2 blocks/set, S=2 sets
Address trace (reads, one byte per read):

0	**[00$\underline{0}$0$_2$],**	miss
1	**[00$\underline{0}$1$_2$],**	hit
7	**[01$\underline{1}$1$_2$],**	miss
8	**[10$\underline{0}$0$_2$],**	miss
0	**[00$\underline{0}$0$_2$]**	hit

	v	Tag	Block
Set 0	1	00	M[0-1]
	1	10	M[8-9]
Set 1	1	01	M[6-7]
	0	?	?

- Allow multiple blocks to be cached in each set:
  - Number of blocks in each set is called "set associativity"
  - Improving temporal locality by allowing a choice regarding which blocks get evicted
- But how do we efficiently search two tags and two valid bits in each set?
- And how do we decide which blocks get evicted?

```c
// Author: Alyssa Kelley
// Assignment 8

// Note: Source code from Eric Wills, and I collaborated with
// Anne Glickenhaus and Kristine Stecker in CIS 314 as we worked on
// this project together.

#include <stdio.h>
#include <stdlib.h>

struct Line {
  unsigned char data[4];
  unsigned int tag;
  unsigned char valid;
};

struct Cache {
  struct Line *lines;
  int numLines;
};

unsigned int getOffset(unsigned int address) {
  // 4B blocks, so offset is bits 0-1
  return address & 0x3;
}

unsigned int getSet(unsigned int address) {
  // 16 sets, so offset is bits 2-6
  return (address >> 2) & 0xF;
}

unsigned int getTag(unsigned int address) {
  // Offset and set are 6 bits total, so tag is high-order 26 bits
  return address >> 6;
}


struct Cache* mallocCache(int numLines) {
  // TODO - malloc a pointer to a struct Cache, malloc a pointer to an array
  // of struct Line instances (array length is numLines).  Also initialize
  // valid to 0 for each struct Line.  Return the struct Cache pointer.
  struct Cache* cache = (struct Cache *) malloc(sizeof(struct Cache)); // casting -> pointer
  cache -> numLines = numLines;
  struct Line* lines = (struct Line *) malloc(sizeof(struct Line));
  cache -> lines = lines;
  // to initialize cache so have everything set to 0 to begin with.
  for(int i = 0; i< numLines; i++)
  {
    cache -> lines[i].valid = 0;
    cache -> lines[i].tag = 0;
    for(int j=0; j<4; j++)
    {
      cache -> lines[i].data[j] = 0;
    }
  }
  return cache;
}

void freeCache(struct Cache *cache) {
  free(cache->lines);
  free(cache);
}

void printCache(struct Cache *cache) {
  // TODO - print all valid lines in the cache.
  // This runs when you hit p.
  for(int i = 0; i<cache->numLines; i++) // going through the lines in the cache
  {
    if(cache->lines[i].valid == 1) // making sure it is valid
    {
      printf("set: %d - tag: %d - valid: %d - data:", i, cache->lines[i].tag, cache->lines[i].valid);
      for(int j=0; j<4; j++)
      {
      printf(" %.2x", cache->lines[i].data[j]); // printing the cache data as two lower case hex values if it is valid.
      }
    printf("\n");
  }
}
}
```

```
void readValue(struct Cache *cache, unsigned int address) {
  // TODO - check the cache for a cached byte at the specified address.
  // If found, indicate a hit and print the byte.  If not found, indicate
  // a miss due to either an invalid line (cold miss) or a tag mismatch
  // (conflict miss).

  /* Pass in cache and address and pull off set, tag, and offset from that
  and the point is to go look for this value in the cache
  Direct mapped so we are allowed to look in to get the line
  directly with set on line 98.
  */

  // parse the address
  unsigned int set = getSet(address);
  unsigned int tag = getTag(address); // tag is like a key so you need to make sure the tags (aka keys) match up for it to be a match -- think of this
      list hash tables
  unsigned int offset = getOffset(address);

  // Block we need
  printf("looking for set: %d - tag: %d\n", set, tag);
  struct Line* index = &cache->lines[set]; // Direct map so we go look in here and get the lines directly and then it points to it. No need for malloc
      here.

  // Making sure it is valid
  if(index->valid == 0)
  {
    printf("no valid set found - miss! \n");
  }
  else if(index->tag == tag) // It is valid, and the tags match... it is a hit!
  {
    printf("found set: %d - tag: %d - valid: %u - data: ", set, index->tag, offset, index->valid); // this is the print block
    printf(" %.2x", index->data[offset]); // printing the offset with two lower case hex values.
    printf("\n");
    printf("hit!\n");
  }
  else // it is valid, but it does not match - it is a miss.
  {
    // print block
    printf("found set: %d - tag: %d offset: %u - valid: %d - data: ", set, index->tag, offset, index->valid);
    // print data
    printf(" %.2x", index->data[offset]);
    printf("\n");
    printf("tags don't match - miss!\n");
  }
}

void writeValue(struct Cache *cache, unsigned int address, unsigned char *newData) {
  // Calculate set and tag for address
  unsigned int s = getSet(address);
  unsigned int t = getTag(address);

  // Get pointer to cache line in the specified set
  struct Line *line = &cache->lines[s];

  // Determine if we have a valid line in the cache that does not contain the
  // specified address - we detect this by checking for a tag mismatch
  if (line->valid && line->tag != t) {
    unsigned char *data = line->data;
    printf("evicting line - set: %x - tag: %x - valid: %u - data: %.2x %.2x %.2x %.2x\n",
        s, line->tag, line->valid, data[0], data[1], data[2], data[3]);
  }

  // Copy new data to line (could use memcpy here instead)
  for (int i = 0; i < 4; ++i) {
    line->data[i] = newData[i];
  }

  // Update line tag, mark line as valid
  line->tag = t;
  line->valid = 1;

  printf("wrote set: %x - tag: %x - valid: %u - data: %.2x %.2x %.2x %.2x\n",
      s, line->tag, line->valid, newData[0], newData[1], newData[2], newData[3]);
}
```

```c
int main() {
  struct Cache *cache = mallocCache(16);

  // Loop until user enters 'q'
  char c;
  do {
    printf("Enter 'r' for read, 'w' for write, 'p' to print, 'q' to quit: ");
    scanf(" %c", &c);

    if (c == 'r') {
      printf("Enter 32-bit unsigned hex address: ");

      unsigned int a;
      scanf(" %x", &a);

      readValue(cache, a);
    } else if (c == 'w') {
      printf("Enter 32-bit unsigned hex address: ");

      unsigned int a;
      scanf(" %x", &a);

      printf("Enter 32-bit unsigned hex value: ");

      unsigned int v;
      scanf(" %x", &v);

      // Get byte pointer to v
      unsigned char *data = (unsigned char *)&v;

      writeValue(cache, a, data);
    } else if (c == 'p') {
      printCache(cache);
    }
  } while (c != 'q');

  freeCache(cache);
}
```

Concluding Observations

- Programmer can optimize for cache performance
  - How data structures are organized
  - How data are accessed
    - Nested loop structure
- All systems favor "cache friendly code"
  - Getting absolute optimum performance is very platform specific
    - Cache sizes, block size, associativities, etc.
  - Can get most of the advantage with generic code
    - Keep working set reasonably small (temporal locality)
    - Use small strides (spatial locality)

Assignment 8 - Cache Hits and Misses