CS33 : Introduction to Computer Organization

Jochen Haber
MS4000A
jhaber@cs.ucla.edu

Lecture: MW 4PM-6PM

Office Hours:
MW: 6PM-7PM
BH4532B

Discussion as Enrolled:

Sec 1A PAB1749: F2PM-4PM Garrett Johnston (gjohnston@cs.ucla.edu)
TuTh: 9:30-10:30
Sec 1B BH5272: F4PM-6PM Peng Wei (pengweiprc@ucla.edu)
TuTh: 3:30-4:30
Sec 1C BH3400: F4PM-6PM Brandon Wu (brandonwu@cs.ucla.edu)
MTh: 9:30AM-10:30AM
Sec 1D BH5252: F4PM-6PM Uen-Tao Wang (cerberiga@ucla.edu)
M1:00PM-3:00PM
Sec 1E BH5273: F4PM-6PM Alex Wood (alex.wood@cs.ucla.edu)
MW 11:30AM-12:30AM

Text:
Randal E. Bryant and David R. O'Hallaron
Computer Systems: A Programmer's Perspective (CSPP)
2nd (North American!) Edition, Prentice Hall 2010.

What this course is about

Computer organization:

How does a computer really work?
From a hardware perspective but exposed by software

Course focus:

chemicals and manufacturing:	NO
imbedded architecture	NO, but just an exposure
general architecture	YES, mostly about software
heavy on memory architecture	YES
compilers	YES
enabling software (systems programs)	NO

Outline of course : (CSPP page xxviii Figure 2 “Course” ICS plus concurrent programming):

Topic	CSPP
Introduction	(1)
Tour of systems	(1)
Data representation	(2)
Machine language	(3)
Code optimization	(5)
Memory hierarchy	(6)
Linking	(7)
Exception control flow	(8)
Virtual memory	(9)
Concurrent programming	(12)

Course Details

Course work

18 lectures	36 hours		
9 discussions	18 hours		
2 mid-term exams	open notes:	15 points (each)	30
final exam	open notes:	30 points	30

Outside work 90 hours

4 labs	15 points (each)	60
? homework	only passing grade	+1 point for pass, extra credit

Perfect grade	**120 plus EC homeworks**

Labs/homeworks

Will be tested on SEAS Linux machines.
Collaboration is encouraged but should be symmetrically acknowledged.
Late submissions not accepted.

Exams:

Postponed exams by prior notice only

Course Enrollment Issues

Spreadsheets?

Tentative Lecture Schedule

Lecture	Date	Topic
1	10/6/2014	Introduction/Tour of Systems
2	10/8/2014	Data Representation
3	10/13/2014	Data Representation
4	10/15/2014	Machine Language
5	10/20/2014	Assembly Language
6	10/22/2014	Midterm*
7	10/27/2014	Code Optimization
8	10/29/2014	Code Optimization
9	11/3/2014	Memory Hierarchy
10	11/5/2014	Caching
11	11/10/2014	Caching
12	11/12/2014	Midterm*
13	11/17/2014	Linking
14	11/19/2014	Exception Control Flow
15	11/24/2014	Virtual Memory
16	11/26/2014	Virtual Memory
17	12/1/2014	Concurrent Programming
18	12/3/2014	Concurrent Programming
19	12/8/2014	Review
20	12/10/2014	Review
	12/16/2014	Final Exam 8AM-11AM
	11/28/2014	No Discussion - Thanksgiving Holiday
		* No office hour

What is a Computer?

Abacus?
Cash Register?
Thermometer?
Stereo Amplifier?

Analog (continuous, real)

Transducers
Problem Solvers

Digital (discrete, rational)

Mainframe
Mini
PC
Imbedded

Theoretical (set theory based)

FSM	regular languages
PDA	context free
LBA	context sensitive
TM	recursive

TMHP, Goedel's Incompleteness, Russell's paradox
Relationship to grammars
Chomsky, Greibach

Non-programmable versus programmable

Stored program vs fixed.
Modify its stored program

Abacus

What is a Computer?

Abacus?
Cash Register?
Thermometer?
Stereo Amplifier?

Analog (continuous, real)

- Transducers
- Problem Solvers

Digital (discrete, rational)

- Mainframe
- Mini
- PC
- Imbedded

Theoretical (set theory based)

FSM	regular languages
PDA	context free
LBA	context sensitive
TM	recursive

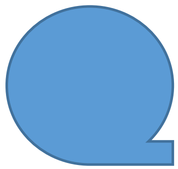
TMHP, Goedel's Incompleteness, Russell's paradox
Relationship to grammars
Chomsky, Greibach

Non-programmable versus programmable

- Stored program vs fixed.
- Modify its stored program

Short Digression on Theoretical Computers

Input Tape

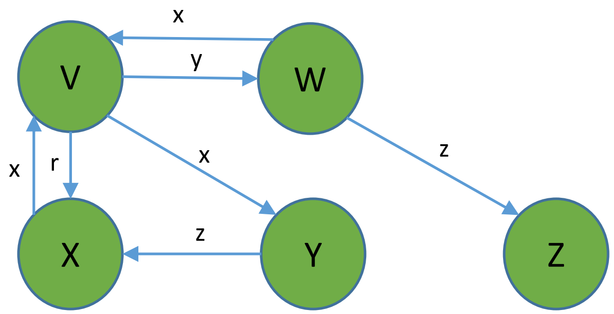


Valid Input Characters: A = { a,b,c,… } = alphabet

A* = set of all possible strings made up of alphabet

Any subset of A* is a “language”

State Diagram



States: S = { V,W,X… } Initial and Final State

Arrows are transitions from state to state depending of what is under the read head of the tape.

Machine starts in initial state with a string on the tape, stops when no further moves are possible.

What is a Computer?

Abacus?
Cash Register?
Thermometer?
Stereo Amplifier?

Analog (continuous, real)

Transducers
Problem Solvers

Digital (discrete, rational)

Mainframe
Mini
PC
Imbedded

Theoretical (set theory based)

FSM	regular languages
PDA	context free
LBA	context sensitive
TM	recursive

TMHP, Goedel's Incompleteness, Russell's paradox
Relationship to grammars
Chomsky, Greibach

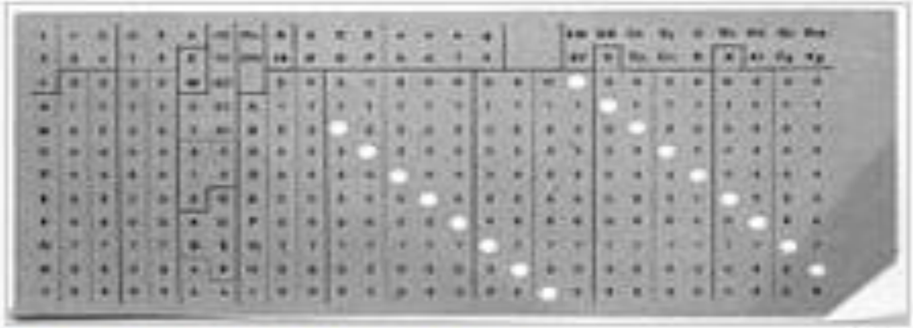
Non-programmable versus programmable

Stored program vs fixed.
Modify its stored program

A Short History

2400 BC	Greek - Abacus
1100 BC	Chinese - Abacus
800 AD	Persians - Analog Machines/Astronomical Calculators
1200	Europe – Logic Machines
1850	Charles Babbage – Differential Calculator
1890	Herman Hollerith – Census Tabulator
1936	Alan Turing – Article on Turing Machines
1936	Konrad Zuse – Programmable Mechanical Machine (Z1)
1939	Hewlett Packard founded
1944	IBM – Howard Mark I
1945	John von Neumann – report on programmable computers
1946	ENIAC – U Penn
1954	WEIZAC – Weizmann Institute
1954	IBM 650 – commercially produced computer
1957	Fortran programming language
1959	COBOL programming language
1964	PL/1 programming language
1964	IBM System 360
1965	DEC PDP-8 mini computer
1965	Packet Switching – basis of Internet UCLA/Stanford
1970	UNIX
1975	Microsoft founded

Punched Cards



Hollerith punched card

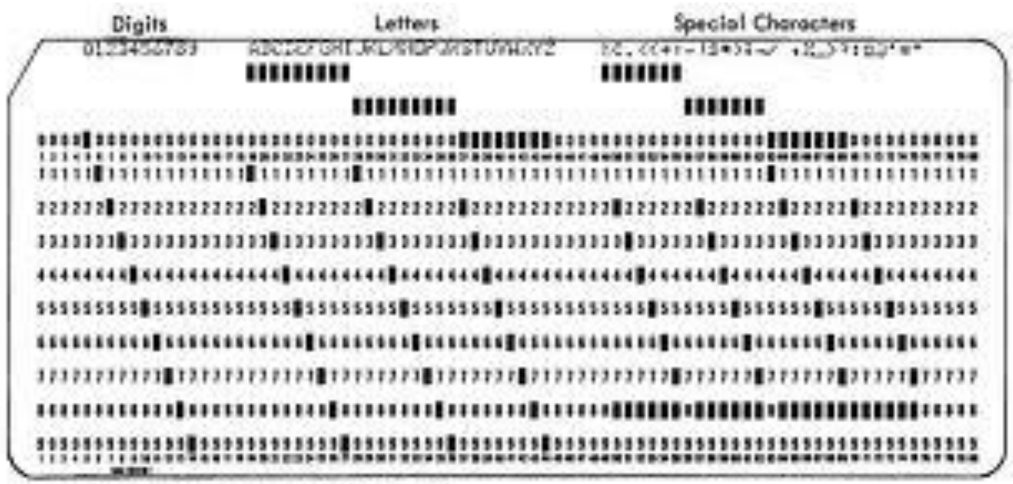
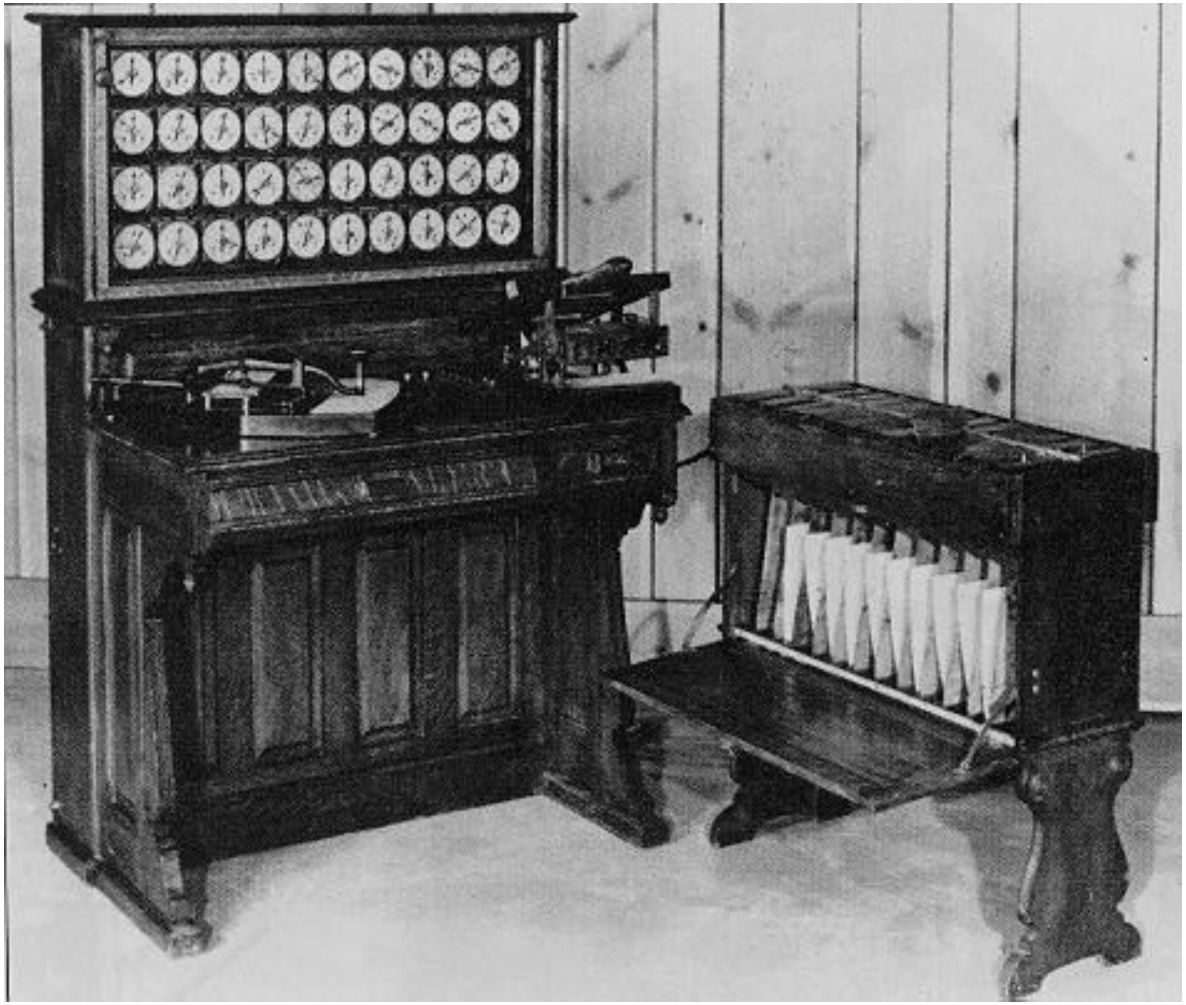


Figure 4. Card Codes and Graphics for 64-Character Set

A Short History

2400 BC	Greek - Abacus
1100 BC	Chinese - Abacus
800 AD	Persians - Analog Machines/Astronomical Calculators
1200	Europe – Logic Machines
1850	Charles Babbage – Differential Calculator
1890	Herman Hollerith – Census Tabulator
1936	Alan Turing – Article on Turing Machines
1936	Konrad Zuse – Programmable Mechanical Machine (Z1)
1939	Hewlett Packard founded
1944	IBM – Howard Mark I
1945	John von Neumann – report on programmable computers
1946	ENIAC – U Penn
1954	WEIZAC – Weizmann Institute
1954	IBM 650 – commercially produced computer
1957	Fortran programming language
1959	COBOL programming language
1964	PL/1 programming language
1964	IBM System 360
1965	DEC PDP-8 mini computer
1965	Packet Switching – basis of Internet UCLA/Stanford
1970	UNIX
1975	Microsoft founded

Hollerith Census Calculator



A Short History

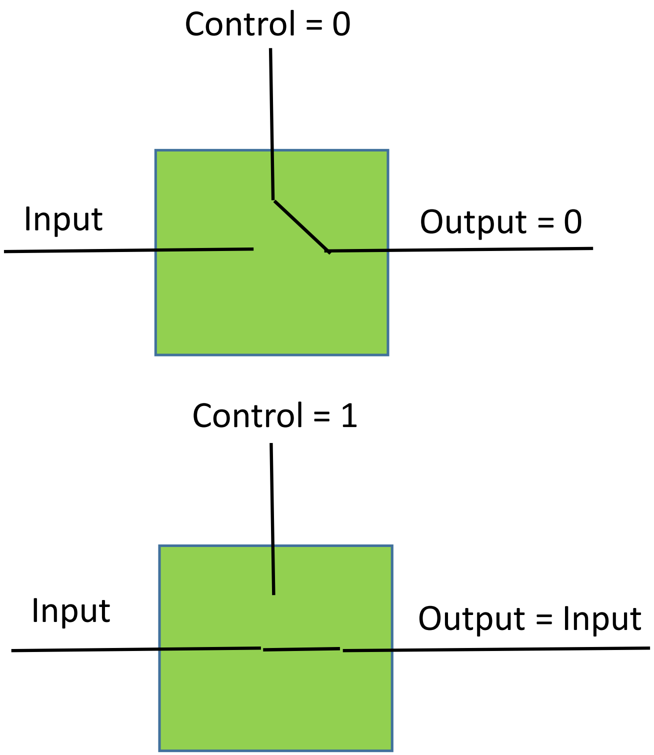
2400 BC	Greek - Abacus
1100 BC	Chinese - Abacus
800 AD	Persians - Analog Machines/Astronomical Calculators
1200	Europe – Logic Machines
1850	Charles Babbage – Differential Calculator
1890	Herman Hollerith – Census Tabulator
1936	Alan Turing – Article on Turing Machines
1936	Konrad Zuse – Programmable Mechanical Machine (Z1)
1939	Hewlett Packard founded
1944	IBM – Howard Mark I
1945	John von Neumann – report on programmable computers
1946	ENIAC – U Penn
1954	WEIZAC – Weizmann Institute
1954	IBM 650 – commercially produced computer
1957	Fortran programming language
1959	COBOL programming language
1964	PL/1 programming language
1964	IBM System 360
1965	DEC PDP-8 mini computer
1965	Packet Switching – basis of Internet UCLA/Stanford
1970	UNIX
1975	Microsoft founded

Babbage Difference Engine

A Short History

2400 BC	Greek - Abacus
1100 BC	Chinese - Abacus
800 AD	Persians - Analog Machines/Astronomical Calculators
1200	Europe – Logic Machines
1850	Charles Babbage – Differential Calculator
1890	Herman Hollerith – Census Tabulator
1936	Alan Turing – Article on Turing Machines
1936	Konrad Zuse – Programmable Mechanical Machine (Z1)
1939	Hewlett Packard founded
1944	IBM – Howard Mark I
1945	John von Neumann – report on programmable computers
1946	ENIAC – U Penn
1954	WEIZAC – Weizmann Institute
1954	IBM 650 – commercially produced computer
1957	Fortran programming language
1959	COBOL programming language
1964	PL/1 programming language
1964	IBM System 360
1965	DEC PDP-8 mini computer
1965	Packet Switching – basis of Internet UCLA/Stanford
1970	UNIX
1975	Microsoft founded

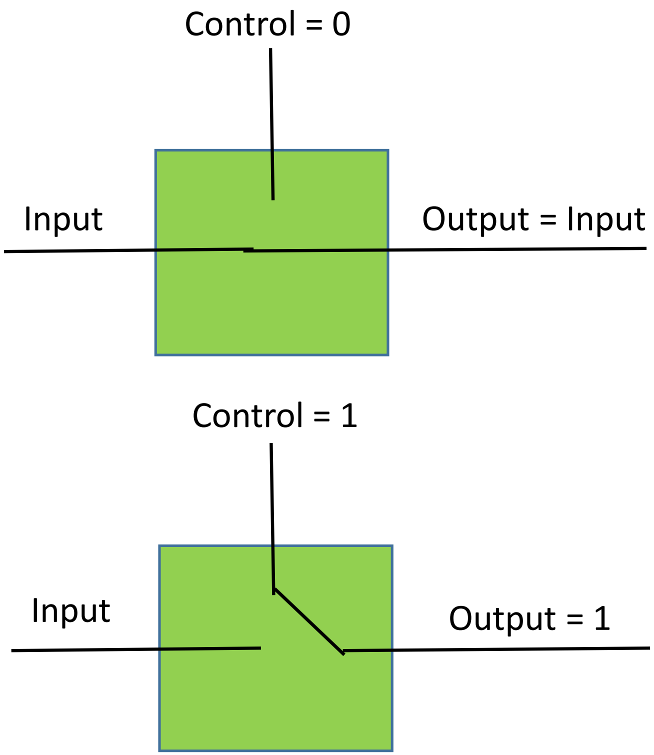
Transistors



And Gate

Input \ Control	0	1
0	0	0
1	0	1

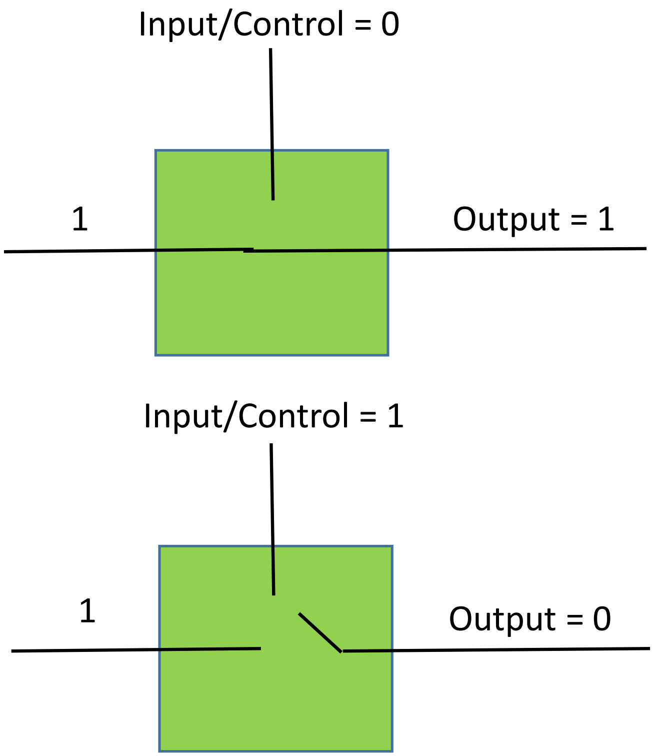
Truth Table



Or Gate

Input \ Control	0	1
0	0	1
1	1	1

Truth Table



Not Gate

Input \ Control	0	1
0	1	
1		0

Truth Table

Transistor Computers

Mainframes

IBM (605, 1401, 7040, 7094, 360, Model 91, 370, z Series)
1954 1959 1961 1962 1964 1968 1970, 1990

NCR, CDC, Univac

Cray, Illiac

Mini Computers

1965 Digital Equipment PDP-8

Hewlett Packard

PCs

1971 Intel

1975 ALTAIR

1976 Apple

1977 Radio Shack TRS-80, Commodore

1981 IBM DOS/IBM PCs

Compaq

Sun

Dell

HP

Transistor Computers

Handheld

- Digital Watches
- Simple calculators
- HP55
- Blackberry
- Smartphones

Builtin

- Cars
- Air Navigation
- Appliances

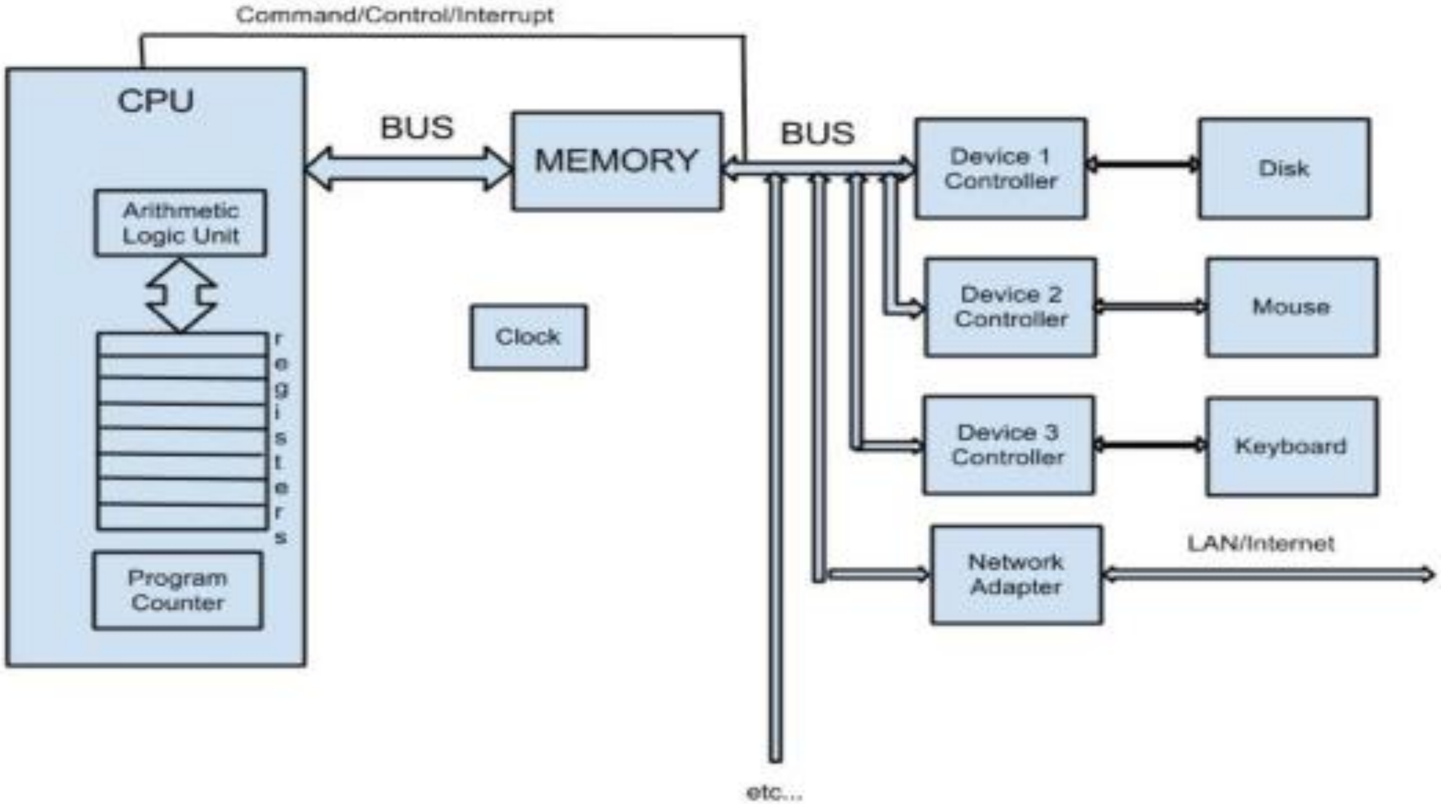
Leftovers from last Lecture

Enrollment situation – new section
Auditors – invite
Kiran Sivakumar
Collaboration -- more to say

This lecture:

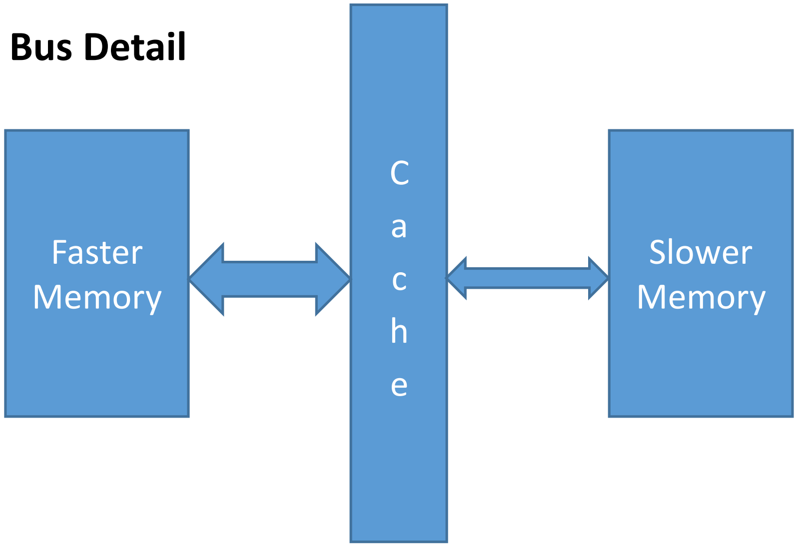
- Computer overview
- Memories
- Main memory
- Binary/hex
- Binary integers
- Integer addition
- Integer multiplication
- Fixed point integer
- Floating point representation
- Boolean Logic/arithmetic

Overview of a Digital Computer



Overview of a Digital Computer

Bus Detail



Memory Combinations:

Registers/Main Memory
ALU/Main Memory
Main Memory/Disk Controller

Main Memory

Huge array of bytes (bits) Each byte has an address. Lets say that x is an int. x has address 0x78,y is a char. Its address is 0x95. z is a short, its address is 0xb4. Memory contains aggregates.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7									x_3	x_2	x_1	x_0				
8																
9						y										
A																
B					z_1	z_0										
C																
D																
E																
F																

IA32: 2^{32}-1 is maximum memory address

X86-64 2^{64}-1 is maximum.

Data Types

Characters: (one byte)

ASCII (Teletype)
EBCDIC

Numbers many different uses, sizes

Integer
Floating point
Decimal?
Pointers

Bit matrices

Black and white image
Color images: RGB

Machine instructions

Op code, operands

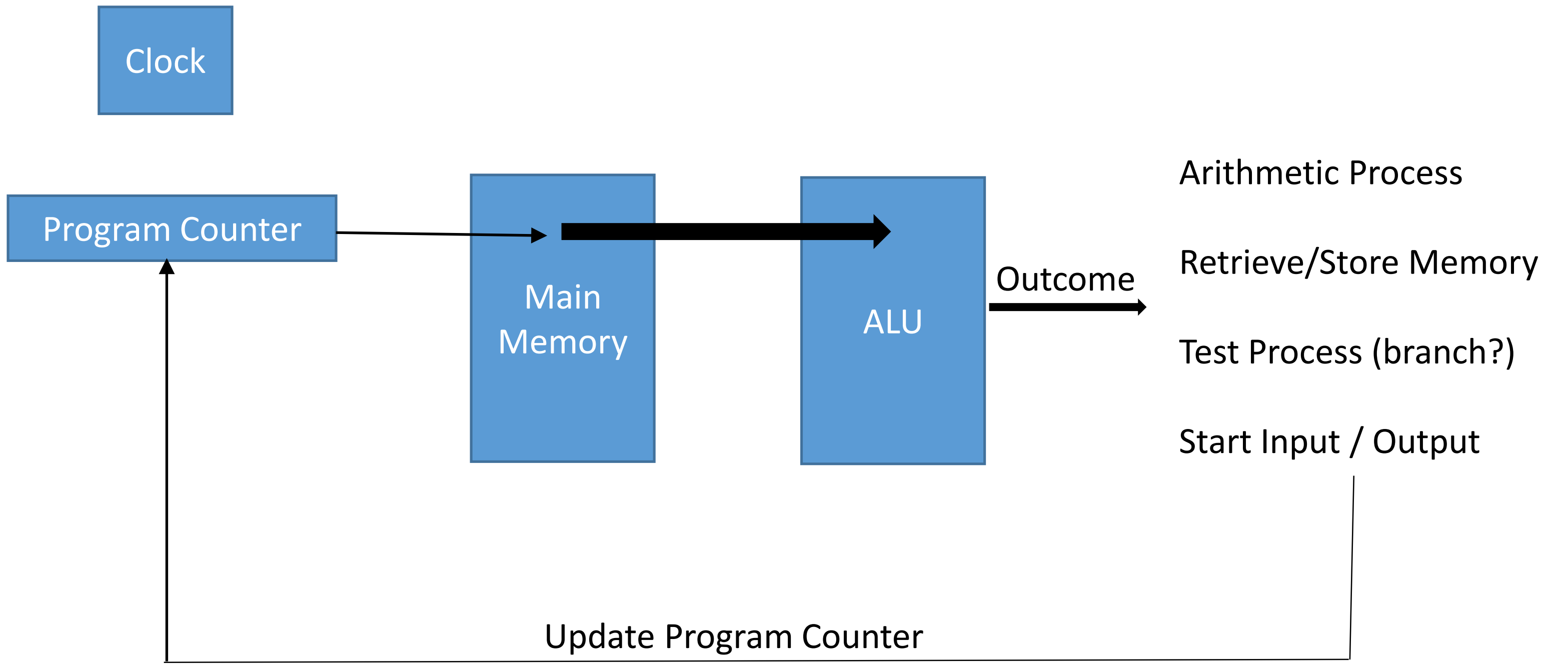
ASCII Code Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

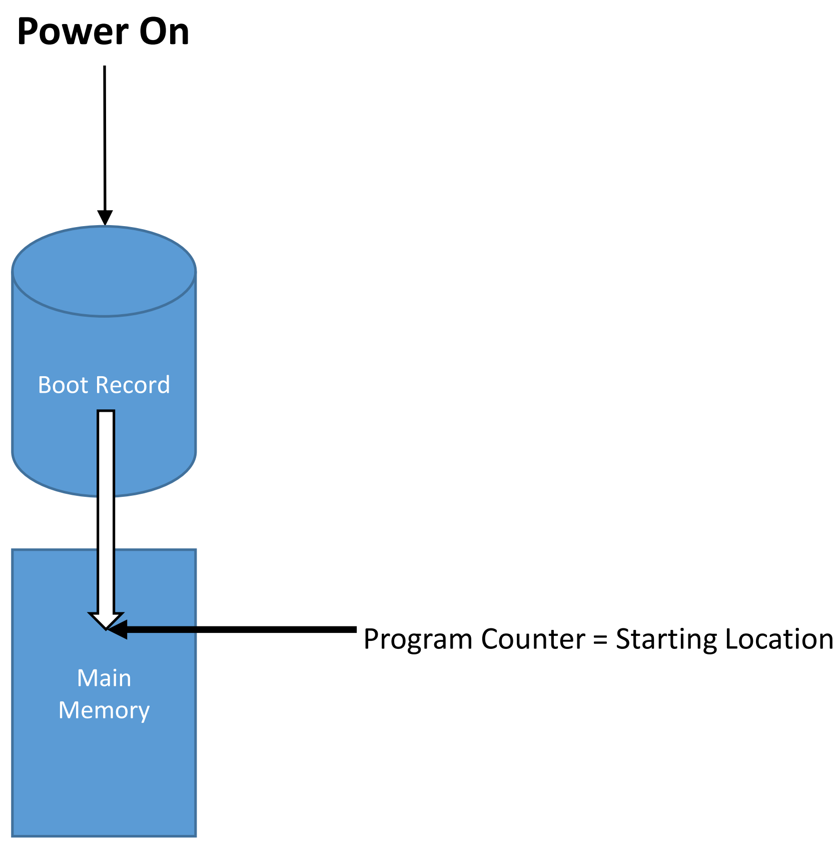
Source: www.LookupTables.com

A = 0x41 = 0100 0001   a = 0x61 = 0110 0001   0 = 0x30, 1 = 0x31, ...

How does it work?

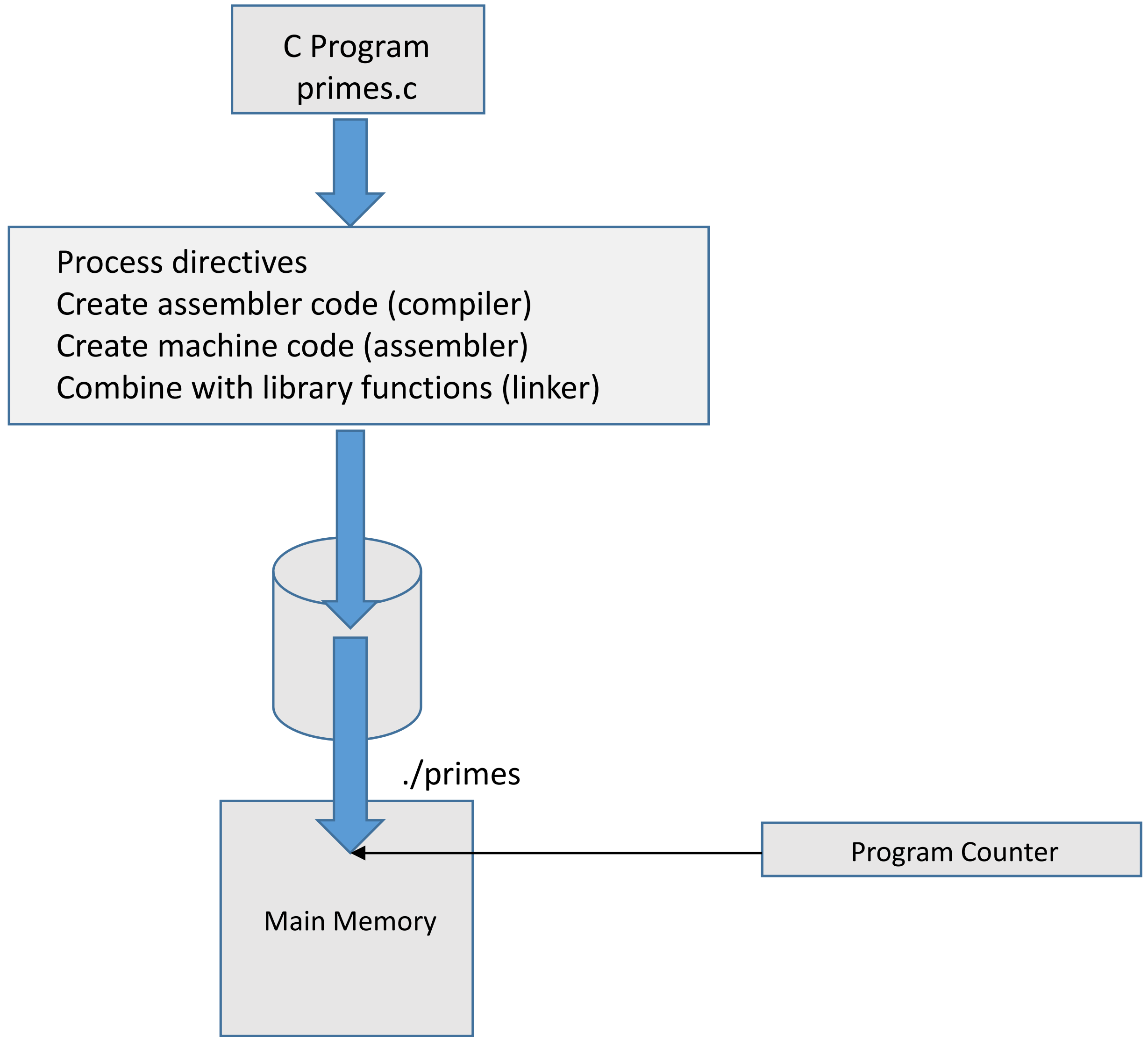


How does it work?



How does it work?

gcc –o primes –lm primes.c



Binary! String of 0's and 1's

01101100
7 6 5 4 3 2 1 0

$$n\ bit\ value = \sum_{i=0}^{n-1} x_i * \ 2^i$$

2^n possible values, -2^{n-1}:2^{n-1} signed

0110 1100
3 2 1 0 3 2 1 0

$$4\ bit\ value = \sum_{i=0}^{3} x_i * \ 2^i$$

16 possible values: 0:15 0-9, A-F

0110 = 2^2 + 2^1 = 6 1100 = 2^3 + 2^2 = 12
6 C

6C
1 0

$$\text{h}exadecimal\ value = \sum_{i=0}^{1} x_i * \ 16^i$$

256 possible values: 0:255, -127:127 **signed**

6C = 6*16 + 12*1 = 108

Signed: most significant bit is the sign

$\underset{7}{1}\underset{6}{1}\underset{5}{1}\underset{4}{0}\underset{3}{1}\underset{2}{1}\underset{1}{0}\underset{0}{0}$

$$\text{n bit signed value} = -x_{n-1} * 2^{n-1} + \sum_{i=0}^{n-2} x_i * 2^i$$

11101100 = $-2^7+2^6+2^5+2^3+2^2$ = -128+64+32+8+4 = -20

Easy way to make negative: reverse bits and add 1 … remember to carry

11101100 becomes 00010011 +00000001 becomes 00010100

Which equals 2^4+2^2 = 16+4 = 20

C Data Types

Type	bytes (X68-64)	mnemonic	description
char	1	b	Character (can also be interpreted as integer)
short	2	w	Short integer
int	4	1	Integer
long int	8	q	Long integer
long long int	8	q	Long integer
char *	8	q	Pointer
float	4	s	Single precision floating point
double	8	d	Double precision floating point
long double	16	t	Extended precision floating point

No bit data type

C Integers

4 sizes: 8, 16, 32 and 64 bits

unsigned/signed

$$\text{n unsigned bit value} = \sum_{i=0}^{n-1} x_i * \ 2^i$$

$$\text{n bit signed value} = -x_{n-1} \ * \ 2^{n-1} + \sum_{i=0}^{n-2} x_i * \ 2^i$$

show binary1

GDB Digression

Command		Effect
	Starting and stopping	
quit		Exit gdb
run		Run your program (give command line arguments here)
kill		Stop your program
	Breakpoints	
break sum		Set breakpoint at entry to function sum
break		*0x8048394 Set breakpoint at address 0x8048394
delete 1		Delete breakpoint 1
delete		Delete all breakpoints
	Execution	
stepi		Execute one instruction
stepi 4		Execute four instructions
nexti		Like stepi, but proceed through function calls
continue		Resume execution
finish		Run until current function returns
	Examining code	
disas		Disassemble current function
disas		sum Disassemble function sum
disas 0x8048397		Disassemble function around address 0x8048397
disas 0x8048394 0x80483a4		Disassemble code within specified address range
print /x		$eip Print program counter in hex
	Examining data	
print $eax		Print contents of %eax in decimal
print /x $eax		Print contents of %eax in hex
print /t $eax		Print contents of %eax in binary
print 0x100		Print decimal representation of 0x100
print /x 555		Print hex representation of 555
print /x ($ebp+8)		Print contents of %ebp plus 8 in hex
print *(int *) 0xfff076b0		Print integer at address 0xfff076b0
print *(int *) ($ebp+8)		Print integer at address %ebp + 8
x/2w 0xfff076b0		Examine two (4-byte) words starting at address 0xfff076b0
x/20b sum		Examine first 20 bytes of function sum
	Useful information	
info frame		Information about current stack frame
info registers		Values of all the registers
help		Get information about gdb s: 8, 16, 32 and 64 bits

GDB Digression

```
void main()
  {
  int i =  123 ;
  int j = -123 ;
  char c[20] = "Hello World" ;
  int k[10] = { 1,2,3,4,5,6,7,8,9,10 } ;
  }
```

Homework Problem

CS33 Fall 2014 Homework 1

Due 6PM Oct 12, 2014

Read and understand the text up to page 41. Alternatively, you can wait to start your homework until after the first lecture. The problem is about the concept of “big endian” and “little endian”, as explained in the text. The assignment is to write a program which has no inputs and its only output is “big” or “little”, depending on whether the host machine that your program is running on uses the “big” or “little” storage method.

The assignment will teach you different ways of examining what is in the computer memory, depending on the data types used.

Estimated solution time: 2 hours.

CS33 Fall 2014 Lab1

Due Oct 19, 2014 6PM

This assignment will help you to understand

1) how to convert from decimal to binary and back and
2) how a computer does addition and multiplication.

The C language does not accommodate a “bit” data type where you can actually input/store/change and output a single bit. For instance, in PL/I, a data type: bit(n) designates a bit string where n is the number of bits in the string. There are mechanisms for accessing any bits in the string (substr and array for example) and a constant type: ‘0'B and ‘1'B exists to set and compare bit values. In this assignment, we will work with integer/binary values so we will emulate the “bit” data type by using the C “int” data type and use only values 0 or 1 for values.

First: You must write two procedures:

void to_binary(int n, int w, int *x, int *o)
and
void from_binary(int *x, int w, int *n)

to_binary shall convert the decimal integer “n” into an array of ints “x” of length “w” where “x” is the binary representation of “n” using zeroes and ones. Negative numbers shold be represented in two’s complement. The error flag “o” shall denote that the integer “n” is larger than can fit into a binary representation of length “w”.

from_binary shall convert a “binary” array of length “w” to an integer “n” whose value is the decimal value of the binary number.

Second: Write a procedure which takes two arrays of “binary” numbers produced by to_binary and adds them together using ***only*** Boolean functions: AND, OR, XOR and NOT and outputs a “binary” array:

```
void adder( int *x, int *y, int *z, int *o, int w )
```

“o” is the overflow flag, indicating that the result will not fit in an array of length “w”, where “w” is the length of the binary arrays.

Lab

You must allow for either or both numbers to be negative according to the rules of 2's complement arithmetic.. Also, the program must set an error flag, but compute the added result if addition results in an overflow. Of course, you can use FOR loops and assignments.

<u>Third:</u> using your adder and conversion routines, write a procedure which takes two numbers and multiplies them using only Boolean functions and assignments. Set an error flag when the multiplication overflows. (This is a bit more difficult than for addition.)

```
void mult( int *x, int *y, int *z, int *o, int w )
```

Overall Note: It does not matter, in to_binary, whether the most or least significant bit comes out in x[0] or x[w-1] just as long as you are consistent in all of your procedures.

Hint: for both the adder and multiplier, go back to your grammar school days and recall how you learned how to do it and emulate that. Also, multiplication tables are not required since multiplying by 0 is 0 and multiplying by 1 is 1 or 0. This may not be the most sophisticated adder and multiplier but it will show you how things work.

Use the lab1.c template. Do not change anything in the main() procedure!

Estimated solution time: 8 hours.

Integer Size Conversion

Convert from smaller to larger

Two ways:

unsigned char   i ; i is 8 bits unsigned
unsigned short j ; j is 16 bits unsigned
j = i ;

assign to unsigned: zero propagation

1100 0001 (193) becomes 0000 0000 1100 0001 (193)
0100 0001 (65) becomes 0000 0000 0100 0001 (65)

---

signed   char   i ; i is 8 bits signed
unsigned short j ; j is 16 bits unsigned
j = i ;

assign to signed: sign propagation

1100 0001 (-63) becomes 1111 1111 1100 0001 (65473)
0100 0001 (65) becomes 0000 0000 0100 0001 (65)

no difference if j is signed

Integer Size Conversion

Convert from larger to smaller

Proceed with caution! Value is truncated

convert x to y (y = x ;) where x is s bits and y is t bits and t < s

for unsigned x must be <= 2^t-1
for signed y, abs(x) must be <= 2^{t-1}-1, x >= -2^{t-1}

signed to unsigned for width w, value changes to 2^w+x if x is < 0

unsigned char j ; j is 8 bits signed
signed short i ; i is 16 bits unsigned
j = i ;

1111 1111 1100 0001 (-63) becomes 1100 0001 (193)
1111 1111 0100 0001 (-191) becomes 0100 0001 (65)

Integer Size Conversion Summary

Examples of type conversion: Smaller to larger: signed propagates first bit, unsigned propagates zero, go to smaller: chop off. Comparisons are 32 bits.

From		To		Compare?	top : from, bottom : to
int	-16777216	uint	4278190080	1	11111111000000000000000000000000 11111111000000000000000000000000
short	-12345	int	-12345	1	1100111111000111 11111111111111111100111111000111
short	-12345	uint	4294954951	1	1100111111000111 11111111111111111100111111000111
short	-12345	ushort	53191	0	1100111111000111 1100111111000111
short	2345	int	2345	1	0000100100101001 00000000000000000000100100101001
short	2345	uint	2345	1	0000100100101001 00000000000000000000100100101001
short	2345	ushort	2345	1	0000100100101001 0000100100101001
uint	40000	int	40000	1	00000000000000001001110001000000 00000000000000001001110001000000

Integer Size Conversion Summary

Examples of type conversion: Smaller to larger: signed propagates first bit, unsigned propagates zero, go to smaller: chop off. Comparisons are 32 bits.

From		To		Compare?	top : from, bottom : to
uint	40000	short	-25536	0	00000000000000001001110001000000 1001110001000000
uint	40000	ushort	40000	1	00000000000000001001110001000000 1001110001000000
ushort	53191	int	53191	1	1100111111000111 00000000000000001100111111000111
ushort	53191	short	-12345	0	1100111111000111 1100111111000111
ushort	53191	uint	53191	1	1100111111000111 00000000000000001100111111000111
uint	118727	short	-12345	0	00000000000000011100111111000111 1100111111000111
uint	118727	ushort	53191	0	00000000000000011100111111000111 1100111111000111
uint	4278190080	int	-16777216	1	11111111000000000000000000000000 11111111000000000000000000000000

Integer Size Conversion Summary

Examples of type conversion: Smaller to larger: signed propagates first bit, unsigned propagates zero, go to smaller: chop off. Comparisons are 32 bits.

From		To		Compare?	top : from, bottom : to
uint	4278190080	short	0	0	11111111000000000000000000000000 0000000000000000
uint	4278190081	short	1	0	11111111000000000000000000000001 0000000000000001
uint	4294954951	int	-12345	1	11111111111111111100111111000111 11111111111111111100111111000111

Programmatic Comparison Summary

```
int main()
 {                                                   Bit pattern assigned
 unsigned short k =  53191 ;                              1100111111000111
            short l = -12345 ;                            1100111111000111
 unsigned  int   m =  k ;        00000000000000001100111111000111
           int   n =  k ;        00000000000000001100111111000111
 unsigned  int   o =  l ;        11111111111111111100111111000111
           int   p =  l ;        11111111111111111100111111000111

 printf( "k l ushort short  %d\n", k == l ) ;     movzwl -0x14(%rbp),%edx   k
                                                  movswl -0x12(%rbp),%eax   l
       compare: 0                                 cmp    %eax,%edx

 printf( "k m ushort uint   %d\n", k == m ) ;  movzwl -0x14(%rbp),%eax   k
                                                  cmp    -0x10(%rbp),%eax    m
       compare: 1

 printf( "k n ushort int    %d\n", k == n ) ;     movzwl -0x14(%rbp),%eax   k
                                                  cmp    -0xc(%rbp),%eax     n
       compare: 1

 printf( "k o ushort uint   %d\n", k == o ) ;     movzwl -0x14(%rbp),%eax   k
                                                  cmp    -0x8(%rbp),%eax     o
       compare: 0

 printf( "k p ushort int    %d\n", k == p ) ;     movzwl -0x14(%rbp),%eax   k
                                                  cmp    -0x4(%rbp),%eax     p
       compare: 0
```

Programmatic Comparison Summary

```
printf( "l m short  uint   %d\n", l == m ) ;     movswl -0x12(%rbp),%eax   l
                                               cmp    -0x10(%rbp),%eax   m
     compare: 0

printf( "l n short  int    %d\n", l == n ) ;     movswl -0x12(%rbp),%eax   l
                                               cmp    -0xc(%rbp),%eax    n
     compare: 0

printf( "l o short  uint   %d\n", l == o ) ;     movswl -0x12(%rbp),%eax   l
                                               cmp    -0x8(%rbp),%eax    o
     compare: 1

printf( "l p short  int    %d\n", l == p ) ;     movswl -0x12(%rbp),%eax   l
                                               cmp    -0x4(%rbp),%eax    p
     compare: 1

printf( "m n uint   int    %d\n", m == n ) ;     mov    -0xc(%rbp),%eax    m
                                               cmp    -0x10(%rbp),%eax   n
     compare: 1

printf( "m o uint   uint   %d\n", m == o ) ;     mov    -0x10(%rbp),%eax   m
                                               cmp    -0x8(%rbp),%eax    o
     compare: 0

printf( "m p uint   int    %d\n", m == p ) ;     mov    -0x4(%rbp),%eax    m
                                               cmp    -0x10(%rbp),%eax   p
     compare: 0
}
```

Programmatic Comparison Summary

```
printf( "n o int    uint   %d\n", n == o ) ;        mov   -0xc(%rbp),%eax    n
                                                    cmp   -0x8(%rbp),%eax    o
        compare: 0

 printf( "n p int    int    %d\n", n == p ) ;       mov   -0xc(%rbp),%eax    n
                                                    cmp   -0x4(%rbp),%eax    p
        compare: 0

 printf( "o p uint   int    %d\n", o == p ) ;       mov   -0x4(%rbp),%eax    p
                                                    cmp   -0x8(%rbp),%eax    o
        compare: 1
 }
```

When moving from a word (16 bits) to any kind of long (32 bits) (signed or unsigned), sign propagation occurs when the short is signed.

Zero propagation occurs when the short is unsigned.

When comparing a short to a long, it first converts the short to a long with the same sign propagation rules.

Note that when two shorts are compared, it converts them both to longs (k == l).

Binary Addition

Unsigned

Adding x and y when they are w bits long: Size of x and y : 0 to $2^w - 1$

$2*(2^w - 1) = (2^{w+1} - 2)$ is w+1 bits: overflow!

Let's set w = 12 for example

	11	10	9	8	7	6	5	4	3	2	1	0	
x	0	0	0	0	0	1	1	1	1	0	1	1	123
y	0	0	0	1	0	1	0	0	0	0	0	1	321
sum	0	0	0	1	1	0	1	1	1	1	0	0	444
carry	0	0	0	0	0	1	0	0	0	0	1	1	

How to do it?

$\mathrm{sum_i} = \mathrm{mod}(x_i + y_i + \mathrm{carry_{i-1}}, 2)$

$\mathrm{carry_i}$ = floor($(x_i + y_i + \mathrm{carry_{i-1}})$/2,1)

$\mathrm{carry_{i-1}}$ = 0 when i-1 < 0

Binary Addition

Unsigned

let's set w = 8 for example,

	7	6	5	4	3	2	1	0	
x	1	1	1	0	1	0	1	0	234
y	1	1	1	0	1	0	1	0	234
sum	1	1	0	1	0	1	0	0	212
carry	1	1	1	0	1	0	1	0	

What is wrong with this picture?

What is 2^w-1 ? 255. 234+234 = 468 $>$ 255.

$carry_7$= 1 ! Overflow!

The actual wrong answer is 468 - 2^w because we lost a bit.

Binary Addition

Signed

Same as unsigned except:

Adding x and y when they are w bits long: Size of x and y : -2^{w-1} to $2^{w-1}-1$

$2*$ -2^{w-1} = -2^{w} is w bits: overflow!

	11	10	9	8	7	6	5	4	3	2	1	0	
x	0	0	0	0	0	0	0	0	0	0	0	0	0
y	1	1	1	1	1	1	1	1	1	1	1	1	-1
sum	1	1	1	1	1	1	1	1	1	1	1	1	-1
carry	0	0	0	0	0	0	0	0	0	0	0	0	

How to do it?

$\mathrm{sum_i} = \mathrm{mod}(\ x_i + \ y_i + \ \mathrm{carry_{i-1}}, 2\ )$

$\mathrm{carry_i}$ = floor($(x_i + \ y_i + \ \mathrm{carry_{i-1}})$/2,1)

$\mathrm{carry_{i-1}}$ = 0 when i-1 < 0

Binary Addition

show binary2

Binary Multiplication

How do we do *decimal* multiplication?

123 * 321 =

123 * (3 * 10^2 + 2 * 10^1 + 1 * 10^0) =

123 * 3 * 100 + 123 * 2 * 10 + 123 * 1 =

12300 * 3 + 1230 * 2 + 123 * 1 =

36900 + 2460 + 123 = 39483

$$\begin{array}{r} 123 \\ \underline{x321} \\ 123 \\ 2460 \\ \underline{+36900} \\ 39483 \end{array}$$

Binary Multiplication

How do we do binary multiplication: unsigned?

123 * 321 = 0111 1011 * 1 0100 0001

0111 1011 * (1 * 2^8 + 1 * 2^6 + 1 * 2^0 + (lots of times zero terms)) =

0111 1011 0000 0000 + 01 1110 1100 0000 + 0111 1011 =

1001 1010 0011 1011

```
            0111 1011
        x 1 0100 0001
        -------------
            0111 1011
    01 1110 1100 0000
  0111 1011 0000 0000
  -------------------
  1001 1010 0011 1011    (39483)
```

Overflow problems?

x * y : $\log_2$ x + $\log_2$ y is the size of the result

Binary Multiplication

How do we do binary multiplication: signed?

If either multiplicand is negative, take the two's complement and proceed

-23 * 21

-23 = 1111 1110 1001

Two's complement = 0000 0001 0111 = 23

23 * 21 = 0001 0111 * 0001 0101

0001 0111 $* (1 * 2^4 + 1 * 2^2 + 1 * 2^0$ + (lots of zero terms) =

0001 0111 0000 + 0000 0101 1100 + 0000 0001 0111 =

1001 1010 0011 1011

```
          0001 0111
     x    0001 0101
          ---------
          0001 0111
          0101 1100
     0001 0111 0000
     --------------
     0001 1110 0011  (483)
```

But must take two's complement of result: 1110 0001 1101 (-483)

Overflow problems?

show binary3

Lecture 4 Prelims

Homework/lab submissions

Oct 22, 2104 CS33 Mid Term Topics

Closed book, open notes.

1. Integer Binary data
   - encode/decode
   - size conversion
   - operations
   - fractional fixed point
2. Floating point
   - encode/decode normalized
   - denormalized
   - operations
3. Boolean operations
   - truth tables
4. Assembly Language
   - operands
5. Stack operation

Fractional Binary Numbers

Review decimal

$abc.def_{10} = 10^2a+10^1b+10^0c+10^{-1}d+10^{-2}e+10^{--3}f$

$123.321 = 1*10^2+2*10^1+3*10^0+3*10^{-1}+2*10^{-2}+1*10^{-3}$

Could do the same in binary

$abc.def_2 = 2^2a+2^1b+2^0c+2^{-1}d+2^{-2}e+2^{--3}f$

$101.101_2 = 2^21+2^10+2^01+2^{-1}1+2^{-2}0+2^{--3}1 =$

$4+0+1+.5+0+0.125 = 5.625_{10}$

Multiply by 2

$2^31+2^20+2^11+2^01+2^{-1}0+2^{-2}1 = 8+0+2+1+0+0.25 = 11.25_{10}$

Show binary4

Problems:

Where is the binary point?
Arithmetic
Limited range of values
Rounding ?
- to even
- towards zero
- down
- up

Floating Point Binary Numbers

Create "floating binary point" designated as part of the binary string

Several floating point formats IEEE-754 most accepted.
IBM "excess 64 hexadecimal"

A p bit floating point number has 3 elements: sign s: 1 bit , significand M (fractional part): n bits, exponent E: p-n-1 = k bits

M and E are n and k bit unsigned binary numbers

The value represented is $V = -1^{s} \text{ x M x } 2^{E}$

s	e	e	..	..	E	f	f	..	..	f
	k-1	k-2			0	n-1	n-2			0
p	p-1	p-2	..	..	p-k	p-k-1	p-k-2	..	..	0

Floating Point Binary Numbers

Several floating point formats **IEEE-754** most accepted.
IBM "excess 64 hexadecimal"

Normalized:

e is not all zeroes and not all ones. i.e. 0 > e < 2^k-1

E = e – Bias where bias $= 2^{k-1} - 1$ and e = $e_{k-1}e_{k-2}$ ... e_0

M = 1+f where f = $f_{n-1}f_{n-2}$ ... f_0 (note that M = 1.f in binary and the 1 is implied in f)

The value represented is $V = -1^s \text{ x M x } 2^E$

Denormalized:

exponent field is all zeroes

E = 1 – Bias

M = f (no implied 1)

Special Values:

e's are all ones and f's are all zeroes: infinity positive or negative depending on s.

e's are all ones and f's are not all zeroes: invalid value or Not a Number (NaN)

Floating Point Binary Numbers

Focus on single precision: 32 bit: word with bits 31, 30, ... , 1, 0.

	7	6	5	4	3	2	1	0	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	e	e	e	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m	m

Bit 31 = s = 1 when negative, 0 when positive: s or sign

Bits 30-23 = 8 (k) bit unsigned integer “excess 127": E or exponent

Subtract 127 (bias $2^8 - 1$) from actual value to get the value of e

Values 255 (infinity) and 0 (denormalized number) have reserved meaning

Bits 22-0 = 23 (m) bit unsigned fractional number: significand, mantissa or fraction:

Mantissa is actually 24 bits, supplemented by a “phantom” bit which is always 1, as if there were 24 bits 23-0 and bit 23 is 1 interpreted as:

$M = 1.b_{22}b_{21} \ldots b_1b_0$

which is equal to $1+2^{-1}b_{22}+2^{-2}b_{21}+ \ldots +2^{-22}b_1+2^{-23}b_0$

2^e multiplies M (shifts the binary point + = right, - = left), attach the sign value $-1^s\ 2^e\ M$

show binary5

For double precision k = 12, m = 52

Floating Point Binary Numbers

Example for 16 bit: word with bits 15, ... , 1, 0. 5 bit sign

1 5	1 4	1 3	1 2	1 1	1 0	9	8	7	6	5	4	3	2	1	0
	4	3	2	1	0	9	8	7	6	5	4	3	2	1	0
s	e	e	e	e	e	m	m	m	m	m	m	m	m	m	m

Bias = $2^4 - 1 = 15$ for normalized: 0 > exponent < 31 (00001 to 11110), -14 > E < 15

Consider bits 9-0 as a 10 bit unsigned binary number: f. Remember M = 1.f or m = 1+f/1024

Example: 0 10001 10 1100 000 normalized!

exponent = 17 so E = 2 (17-15)

f = 512+128+64 = 704 so the fractional part of f is 512/1024+128/1024+64/1024 = 11/16

M = 1.6875 (phantom) = 16/16 + 11/16 = 27/16

V = 4 x 27/16 = 27/4 = 6.75

Floating Point Binary Numbers

Example: 0 00000 10 1100 000 Denormalized!

exponent = 0 so E = 1-bias = -14

f = 512+128+64 = 704 so the fractional part of f is 512/1024+128/1024+64/1024 = 11/16

M = .6875 due to Denormalized rule (no phantom)

V = 2^{-14} x 11/16 = 11/2^{10} = 11/1024

Example: 0 11111 00 0000 0000 +Infinity!
1 11111 00 0000 0000 -Infinity!

Example: 0 11111 00 1000 0000 Nan (Not a Number)

Floating Point Binary Addition

Focus on single precision: 32 bit: word with bits 31, 30, ... , 1, 0.

How do we add decimal? Pretend we have 7 digit limit on number of digits. Align the decimal points and add.

123 + .321 = 123.0000 + 000.3210

123.0000
000.3210

1234 + .321 = 1234.000 + 0000.321

1234.000
0000.321

12345 + .321 = 12345.00 = 00000.32

12345.00
00000.321 becomes

12345.00
00000.32 due to 7 digit limit for A + B, abs($\log_{10}A-\log_{10}B$) digits lost from smaller addend

Same for binary. Align the binary point and add.

for A + B, abs($\log_2 A-\log_2 B$) bits lost from smaller addend

show binary6

Floating Point Binary Multiplication

Focus on single precision: 32 bit: word with bits 31, 30, ... , 1, 0.

How do we multiply decimal? Pretend we have 7 digit limit on number of digits. Align the decimal points, multiply and move the decimal point

10 * .01 = 10.00 * 00.01

Binary: multiply the mantissas and add the exponents

Sum of exponents, e: must be <= 254, >= -126

show binary7

Due to rounding problems, Floating point arithmetic is NOT associative

(A+B) +C not necessarily = A + (B+C)

If A and B have grossly different logs but B an C the same.

(12345 + .321) + .338 = 12345.32 + .338 = 12345.68

12345 + (.321 + .338) = 12345 + .659 = 12345.66

Floating Point Binary Multiplication

Example of multiplication accuracy

1	1.0000000	3f800000	2.000000e+00	40000000
1	0.1000000	3dcccccd	1.100000e+00	3f8ccccd
1	0.0100000	3c23d70a	1.010000e+00	3f8147ae
1	0.0010000	3a83126e	1.001000e+00	3f8020c5
1	0.0001000	38d1b716	1.000100e+00	3f800347
1	0.0000100	3727c5ab	1.000010e+00	3f800054
1	0.0000010	358637bc	1.000001e+00	3f800008
1	0.0000001	33d6bf93	1.000000e+00	3f800001
10	1.0000000	3f800000	1.100000e+01	41300000
10	0.1000000	3dcccccd	1.010000e+01	4121999a
10	0.0100000	3c23d70a	1.001000e+01	412028f6
10	0.0010000	3a83126e	1.000100e+01	41200419
10	0.0001000	38d1b716	1.000010e+01	41200069
10	0.0000100	3727c5ab	1.000001e+01	4120000a
10	0.0000010	358637bc	1.000000e+01	41200001
10	0.0000001	33d6bf93	1.000000e+01	41200000
100	1.0000000	3f800000	1.010000e+02	42ca0000
100	0.1000000	3dcccccd	1.001000e+02	42c83333
100	0.0100000	3c23d70a	1.000100e+02	42c8051f
100	0.0010000	3a83126e	1.000010e+02	42c80083
100	0.0001000	38d1b716	1.000001e+02	42c8000d
100	0.0000100	3727c5ab	1.000000e+02	42c80001
100	0.0000010	358637bc	1.000000e+02	42c80000
100	0.0000001	33d6bf93	1.000000e+02	42c80000

Floating Point Binary Multiplication

Example of multiplication accuracy

1000	1.0000000	3f800000	1.001000e+03	447a4000
1000	0.1000000	3dcccccd	1.000100e+03	447a0666
1000	0.0100000	3c23d70a	1.000010e+03	447a00a4
1000	0.0010000	3a83126e	1.000001e+03	447a0010
1000	0.0001000	38d1b716	1.000000e+03	447a0002
1000	0.0000100	3727c5ab	1.000000e+03	447a0000
1000	0.0000010	358637bc	1.000000e+03	447a0000
1000	0.0000001	33d6bf93	1.000000e+03	447a0000
10000	1.0000000	3f800000	1.000100e+04	461c4400
10000	0.1000000	3dcccccd	1.000010e+04	461c4066
10000	0.0100000	3c23d70a	1.000001e+04	461c400a
10000	0.0010000	3a83126e	1.000000e+04	461c4001
10000	0.0001000	38d1b716	1.000000e+04	461c4000
10000	0.0000100	3727c5ab	1.000000e+04	461c4000
10000	0.0000010	358637bc	1.000000e+04	461c4000
10000	0.0000001	33d6bf93	1.000000e+04	461c4000
100000	1.0000000	3f800000	1.000010e+05	47c35080
100000	0.1000000	3dcccccd	1.000001e+05	47c3500d
100000	0.0100000	3c23d70a	1.000000e+05	47c35001
100000	0.0010000	3a83126e	1.000000e+05	47c35000
100000	0.0001000	38d1b716	1.000000e+05	47c35000
100000	0.0000100	3727c5ab	1.000000e+05	47c35000
100000	0.0000010	358637bc	1.000000e+05	47c35000
100000	0.0000001	33d6bf93	1.000000e+05	47c35000

Floating Point Binary Multiplication

Example of multiplication accuracy

```
 1000000     1.0000000  3f800000 1.000001e+06  49742410
 1000000     0.1000000  3dcccccd 1.000000e+06  49742402
 1000000     0.0100000  3c23d70a 1.000000e+06  49742400
 1000000     0.0010000  3a83126e 1.000000e+06  49742400
 1000000     0.0001000  38d1b716 1.000000e+06  49742400
 1000000     0.0000100  3727c5ab 1.000000e+06  49742400
 1000000     0.0000010  358637bc 1.000000e+06  49742400
 1000000     0.0000001  33d6bf93 1.000000e+06  49742400
10000000     1.0000000  3f800000 1.000000e+07  4b189681
10000000     0.1000000  3dcccccd 1.000000e+07  4b189680
10000000     0.0100000  3c23d70a 1.000000e+07  4b189680
10000000     0.0010000  3a83126e 1.000000e+07  4b189680
10000000     0.0001000  38d1b716 1.000000e+07  4b189680
10000000     0.0000100  3727c5ab 1.000000e+07  4b189680
10000000     0.0000010  358637bc 1.000000e+07  4b189680
10000000     0.0000001  33d6bf93 1.000000e+07  4b189680
```

Boolean Operations

C allows "Boolean" (George Boole, 1847: Mathematical Analysis of Logic) operations on any data types.

Bitwise or logical

Truth Tables

AND &, &&

	0	1
0	0	0
1	0	1

OR |, ||

	0	1
0	0	1
1	1	1

Exclusive OR ^

	0	1
0	0	1
1	1	0

NOT ~, !

0	1
1	0

Examples

char a = 0x23 ;	00100011 in binary
char b = 0xc5 ;	11000101

Bitwise			Logical		
char c = a & b ;	00000001	0x01	c = a && b ;	00000001	0x01
c = a | b ;	11100111	0xe7	c = a || b ;	00000001	0x01
c = a ^ b ;	11100110	0xe6	c = a ^^ b ;	00000000	0x00
c = ~a ;	11011100	0xdc	c = !a ;	00000000	0x00

Boolean Operator Precedence

~ is unary.

Precedence:

& then

^ then

|

Same precedence, left to right

Example: A | B & C (A | B) & C with A = 1, B = 0, C = 0

1 | 0 & 0	1 | 0 & 0
1 | 0	1 & 0
1	0

Boolean Operator Precedence

Truth Tables

A | B & C

A	B	C	B&C	A|B&C
0	0	0	0	0
0	0	1	0	0
0	1	0	0	0
0	1	1	1	1
1	0	0	0	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

(A | B) & C

A	B	C	A | B	(A | B) & C
0	0	0	0	0
0	0	1	0	0
0	1	0	1	0
0	1	1	1	1
1	0	0	1	0
1	0	1	1	1
1	1	0	1	0
1	1	1	1	1

Laws of Boolean Operations

A & (B | C) = (A & B) | (B & C) distributivity

A | (B & C) = (A | B) & (B | C) associativity

A & B = B & A reflexivity

A | B = B | A reflexivity

~(A & B) = ~A | ~B
~(A | B) = ~A & ~B Augustus De Morgan's laws

lots of other rules: similar to arithmetic, set theory

Laws of Boolean Operations

Notice that A ^ B = (A | B) & ~(A & B)

Truth Table

A	B	A | B	A & B	~ (A & B)	A ^ B
0	0	0	0	1	0
0	1	1	0	1	1
1	0	1	0	1	1
1	1	1	1	0	0

Machine Representation

Programs are made up of machine instructions

Instructions primarily deal with registers, memory

X86-64 Registers: (over IA32: added 32 bits, added general purpose, expanded FP registers)

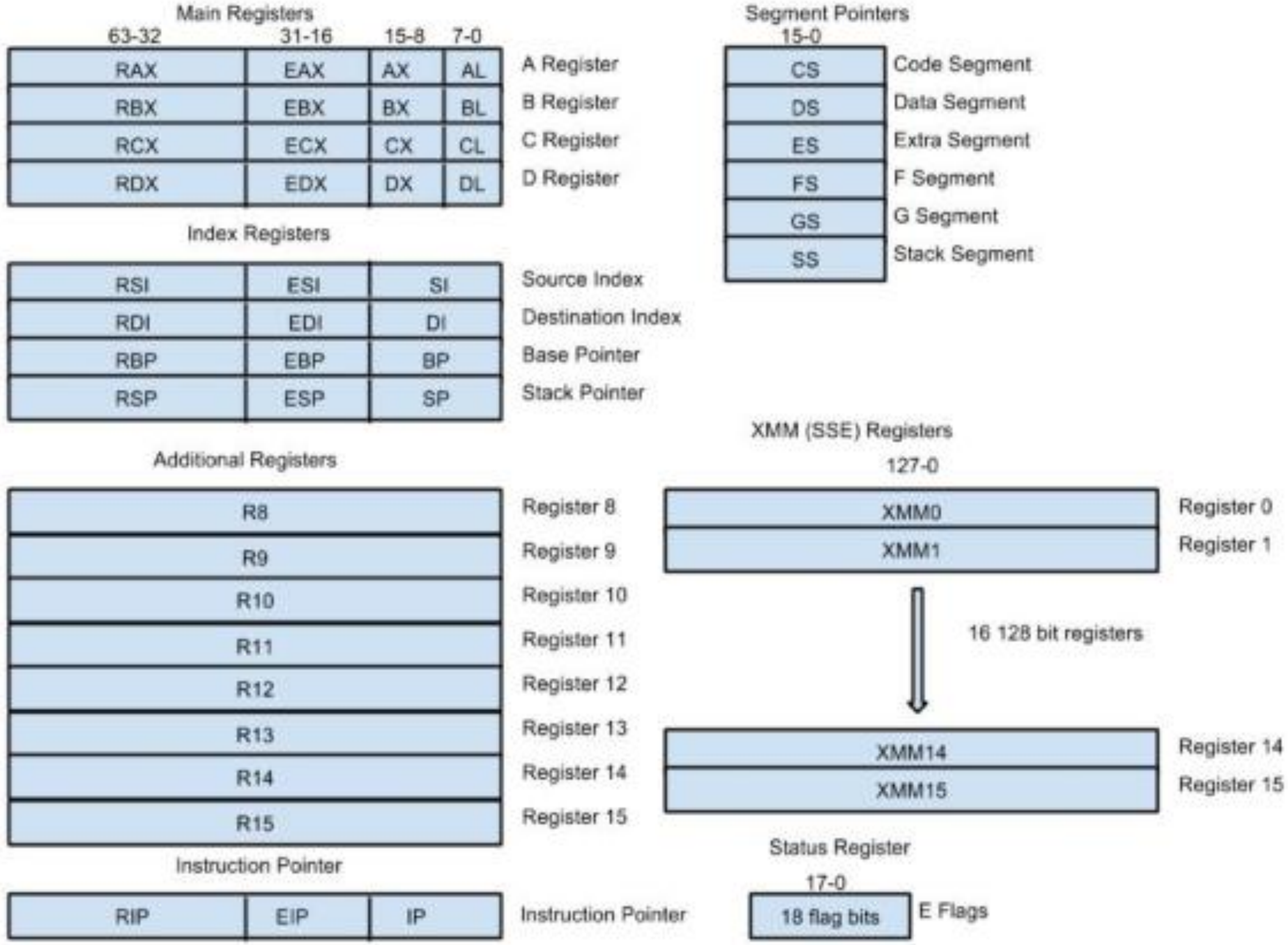
4 main registers
4 index registers
8 additional registers
instruction pointer
6 segment pointers
16 floating point registers
status register

Memory, an array of 2^n bytes, addressed from 0 to $2^n - 1$

Aggregates come out of memory all at once.

Machine Representation

X86-64 registers



Machine Representation

Main Memory

Huge array of bytes (bits) Each byte has an address. Lets say that x is an int. x has address 0x78,y is a char. Its address is 0x95. z is a short, its address is 0xb4.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0																
1																
2																
3																
4																
5																
6																
7									x_3	x_2	x_1	x_0				
8																
9						y										
A																
B					z_1	z_0										
C																
D																
E																
F																

Anything but char considered as an aggregate. Some machines store with least significant on right (big endian), some on the left (little endian). Can retrieve all at once depending on the instruction.

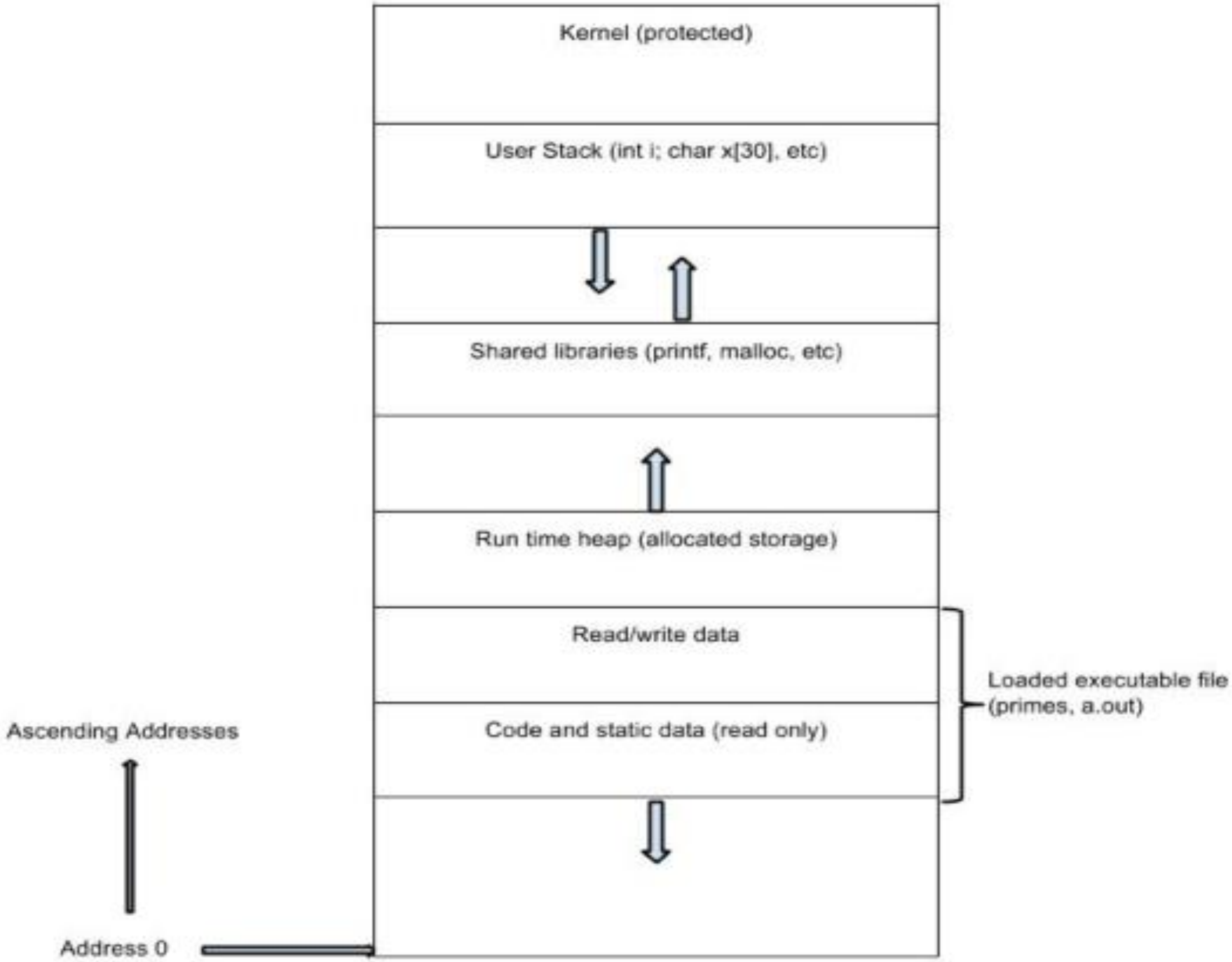
0x12345678 stored as

$[x_3,x_2,x_1,x_0]$ = [0x12,0x34,0x56,0x78] big endian

$[x_3,x_2,x_1,x_0]$ = [0x78,0x56,0x34,0x12] little endian

Machine Representation

Virtual memory: each user seems to have exclusive use of the entire memory. To one user, it looks like:



C Data Types

Type	bytes (X68-64)	mnemonic	description
char	1	b	Character (can also be interpreted as integer)
short	2	w	Short integer
int	4	1	Integer
long int	8	q	Long integer
long long int	8	q	Long integer
char *	8	q	Pointer
float	4	s	Single precision floating point
double	8	d	Double precision floating point
long double	16	t	Extended precision floating point

No bit data type

b = byte (8 bits)
w = word (16 bits)
l = long (32 bits)
q = quad (64 bits)

Transform .c to Machine Language

How does it work from a software point of view?

Program is just a string of operations in memory. Your program is a set of lines in a .txt (.c) file.

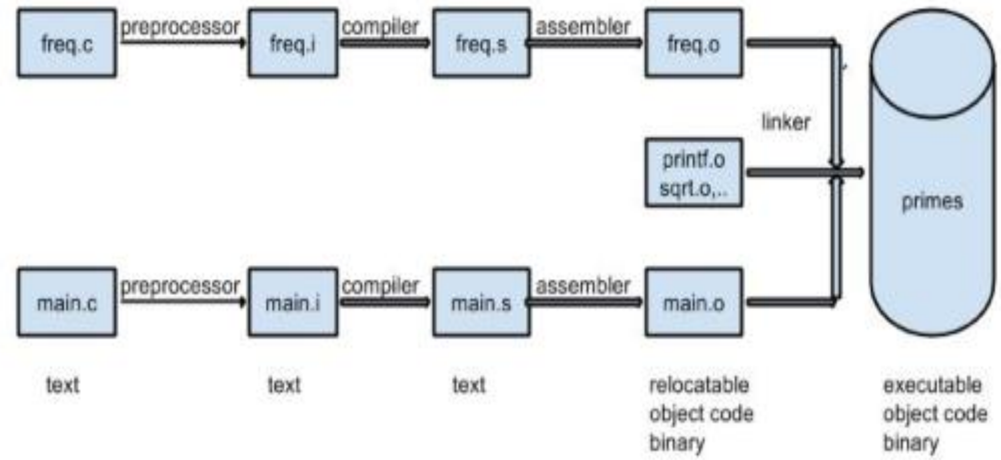
```
freq.c:
#include <math.h>      /* sqrt */
int frequency_of_primes (int n) {
  int i,j;
  int freq= n/2+1 ;
   .
   .
  return freq;
}

main.c:
#include <stdio.h>      /* printf */
#include <time.h>       /* clock_t, clock, CLOCKS_PER_SEC */
int main ()
{
  clock_t t;
  int f,i;

  i = 1000000 ;
  printf ("Calculating... %d ", i );
  t = clock();
  f = frequency_of_primes (i);
  printf ( "%d ",f);
  t = clock() - t;
  printf ("It took me %d clicks (%f seconds).\n",t,((float)t)/CLOCKS_PER_SEC);

  return 0;
}
```

Transform .c to Machine Language

Pre-process, compile, assemble link.

Can stop/start process anywhere, combine any of these files using gcc or g++.
Pre-process, compile, assemble link.

gcc –O1 –lm –o primes freq.c main.c



Book is IA32, lnxsrv at SEAS is X86-64.

How to specify an operand

% implies a register	%rax, %eax, %ax, %al (64) (32) (16) (8)
$ means "immediate" or exactly that value	$0x5: the value 5
parentheses means the value stored at that address memory	(%rax)

Type	Form	Operand value	Name
Immediate	$num	num	Immediate
Register	%rax	%rax	Register
Memory	num	(num)	Absolute
Memory	($rax)	(%rax)	Indirect
Memory	num(%rax)	(num+%rax)	Base+displacement
Memory	(%rax,%rbx)	(%rax+%rbx)	Indexed
Memory	num(%rax,%rbx)	(num+%rax+%rbx)	Indexed
Memory	(,%rax,s)	(%rax*s)	Scaled indexed
Memory	num(,%rax,s)	(num+%rax*s)	Scaled indexed
Memory	(%rax,%rbx,s)	(%rax+%rbx*s)	Scaled indexed
Memory	num(%rax,%rbx,s)	(num+%rax+%rbx*s)	Scaled indexed

Note: s may only be 1, 2, 4 or 8

Operand Examples

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ff	fe	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0
1	fe	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef
2	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee
3	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed
4	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec
5	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb
6	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea
7	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9
8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9	e8
9	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9	e8	e7

Contents of memory on the left,

Assume %rax contains 0x10,
%rbx contains 0x40

Specification	Computation	Address	Value
$0x5	<na>	<na>	5
%rax	<na>	<na>	0x10
0x5	0x05	0x05	0xfa
(%rax)	%rax	0x10	0xfe
0x04(%rax)	0x04+%rax	0x14	0xfa
(%rax,%rbx)	%rax+%rbx	0x50	0xfa
0x04(%rax,%rbx)	0x04+%rax+%rbx	0x54	0xf6
(,%rax,4)	%rax*4	0x40	0xfb
0x05 (,%rax,2)	0x05+%rax*2	0x25	0xf8
(%rax,%rbx,2)	%rax+%rbx*2	0x90	0xf6
0x05 (%rax,%rbx,2)	0x05+%rax+%rbx*2	0x95	0xf1

Assembly Language Instructions

Move instructions: mov source and destination

Combinations of source and destination implied. Proper operation code determined by compiler.

immediate to register	mov	immediate,register
register to register	mov	register,register
memory to register	mov	memory,register
immediate to memory	mov	immediate,memory
register to memory	mov	register,memory

Obviously cannot move to an immediate: mov %rax,$0x40
Cannot move memory to memory

Moving from smaller to larger: sign extension or zero extension

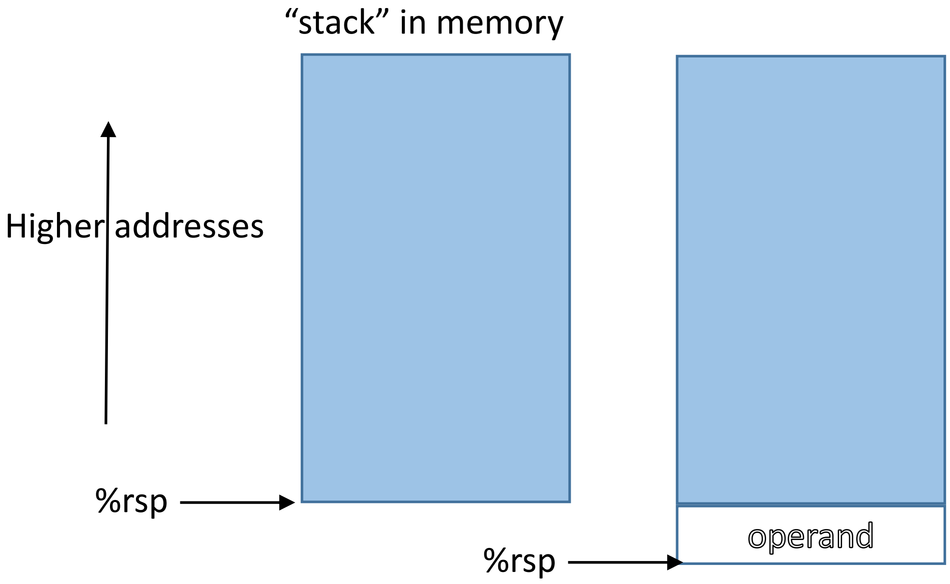
movs	source,dest	sign extension
movz	source,dest	zero extension

Moving from larger to smaller, take only low order.

Move	movb	movw	movl	movq
Move, sign extension	movsbw	movsbl	movswl	movslq
Move, zero extension	movzbw	movzbl	movzwl	movxlq

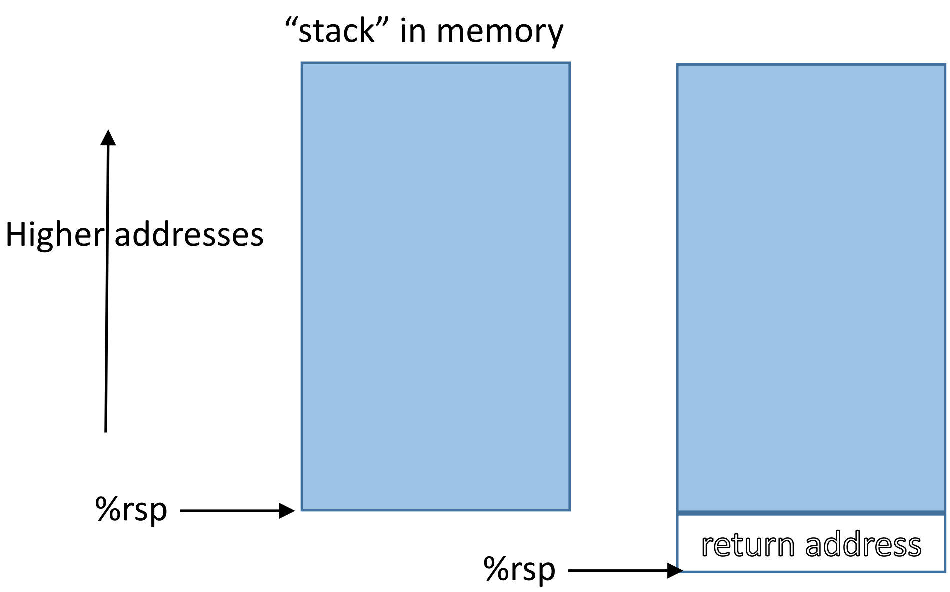
Push, pop instructions: stack manipulation

push	operand			
	short for	%rsp = %rsp – 8 mov operand,(%rsp)	(IA32?)	
pop	operand			
	short for	mov (%rsp),operand %rsp = %rsp + 8		



Call instruction: stack manipulation

```
call    xyz
        short for    %rsp = %rsp – 8             (IA32?)
                     mov   %rip,(%rsp)
                     addq $0x05,(%rsp)
                     mov   address of xyz, %rip
```

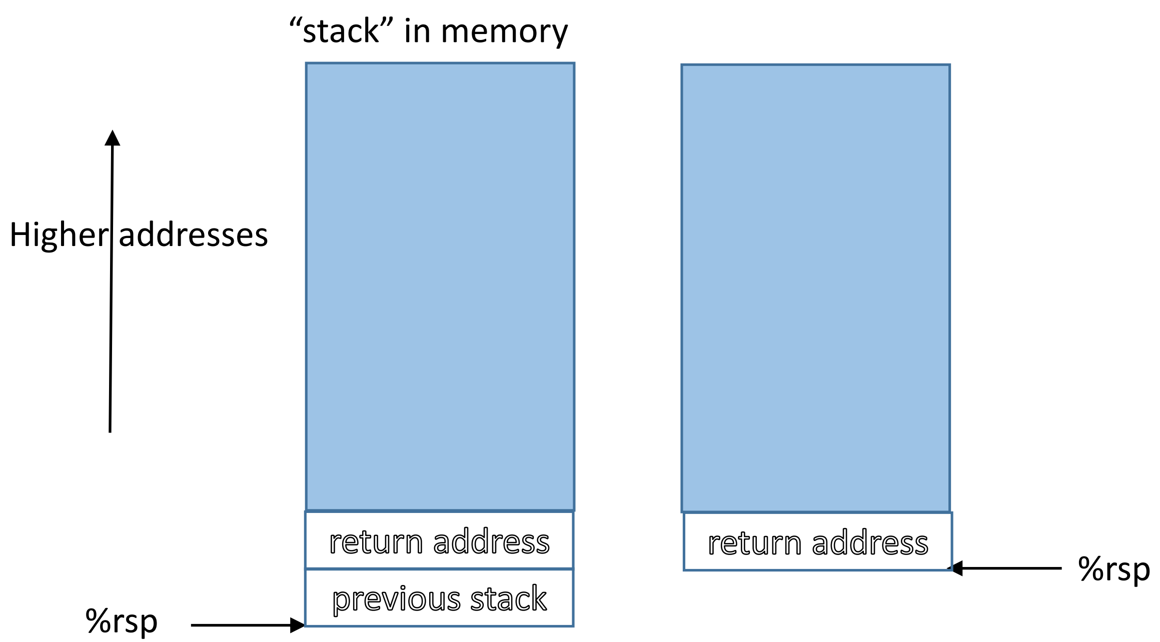


Assembly Language Instructions

Leave instruction: stack manipulation

leave

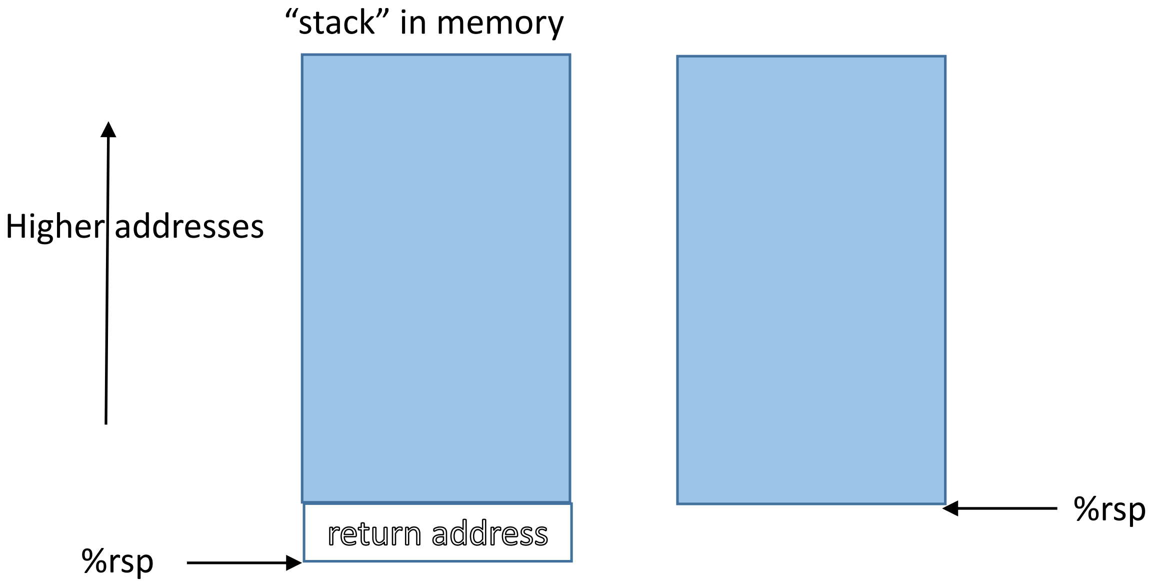
short for mov %rbp,%rsp
mov (%rbp),%rbp
%rsp = %rsp+8



ret instruction: stack manipulation

ret

short for mov (%rsp),%rip
%rsp = %rsp+8



Function call stack manipulation

```
#include <stdio.h>
void to_sub( int *i )
{
int j = 0x78563412 ; char y[10] ;

int k = *i+j ;
other_sub( &j ) ;
printf( "%p\n", &k ) ;
}

void other_sub( int *j )
{
int i ;
i = *j ;
}

int main()
{
int i; char x[30] ;

i = 0x12345678 ;
to_sub( &i ) ;
return 0 ;
}
```

linux interlude stacktest.c

Function call stack manipulation

Dump of assembler code for function main:

```
4004c4 <+0>:    push   %rbp
4004c5 <+1>:    mov    %rsp,%rbp
4004c8 <+4>:    sub    $0x30,%rsp
4004cc <+8>:    movl   $0x12345678,-0x4(%rbp)
4004d3 <+15>:   lea    -0x4(%rbp),%rax
4004d7 <+19>:   mov    %rax,%rdi
4004da <+22>:   callq  0x4004e6 <to_sub>
4004df <+27>:   mov    $0x0,%eax
4004e4 <+32>:   leaveq
4004e5 <+33>:   retq
```

Dump of assembler code for function to_sub:

```
4004e6 <+0>:    push   %rbp
4004e7 <+1>:    mov    %rsp,%rbp
4004ea <+4>:    sub    $0x30,%rsp
4004ee <+8>:    mov    %rdi,-0x28(%rbp)
4004f2 <+12>:   movl   $0x78563412,-0x4(%rbp)
4004f9 <+19>:   mov    -0x28(%rbp),%rax
4004fd <+23>:   mov    (%rax),%edx
4004ff <+25>:   mov    -0x4(%rbp),%eax
400502 <+28>:   lea    (%rdx,%rax,1),%eax
400505 <+31>:   mov    %eax,-0x14(%rbp)
400508 <+34>:   lea    -0x4(%rbp),%rax
40050c <+38>:   mov    %rax,%rdi
40050f <+41>:   callq  0x40052f <other_sub>
400514 <+46>:   mov    $0x400648,%eax
400519 <+51>:   lea    -0x14(%rbp),%rdx
40051d <+55>:   mov    %rdx,%rsi
400520 <+58>:   mov    %rax,%rdi
400523 <+61>:   mov    $0x0,%eax
400528 <+66>:   callq  0x4003b8 <printf@plt>
40052d <+71>:   leaveq
40052e <+72>:   retq
```

Function call stack manipulation

Before Call

```
Breakpoint 1, 4004da in main ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000     0x00000000     0xd08908ed     0x00000032
0x7fffffffe3f0: 0x00000000     0x00000000     0x00400560     0x00000000
0x7fffffffe400: 0x00000000     0x00000000     0x004003a3     0x00000000
0x7fffffffe410: 0xffffe548     0x00007fff     0x004005a5     0x00000000
0x7fffffffe420: 0xd080fba0     0x00000032     0x00400560     0x00000000
0x7fffffffe430: 0x00000000     0x00000000     0x004003e0     0x00000000
0x7fffffffe440: 0xffffe530     0x00007fff     0x00000000     0x12345678
0x7fffffffe450: 0x00000000     0x00000000     0xd081ed1d     0x00000032
```

%rsp

%rbp

rip	rbp	rsp
4da	e450	e420

i

Function call stack manipulation

After Call

```
              4004e6 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe548      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rbp

rip	rbp	rsp
4e6	e450	e418

Return address, %rsp

Function call stack manipulation

After push %rbp

```
                4004e7 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp

%rbp

(%rbp)

rip	rbp	rsp
4e7	e450	e410

Function call stack manipulation

Mov %rsp,%rbp

```
                4004ea in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp

%rbp

rip	rbp	rsp
4ea	e410	e410

Function call stack manipulation

Sub 0x30,%rsp

```
                      4004ee in to_sub ()
              (gdb) x/32w 0x7fffffffe3e0
%rsp  ------->0x7fffffffe3e0: 0x00000000   0x00000000   0xd08908ed   0x00000032
              0x7fffffffe3f0: 0x00000000   0x00000000   0x00400560   0x00000000
              0x7fffffffe400: 0x00000000   0x00000000   0x004003a3   0x00000000
%rbp  ------->0x7fffffffe410: 0xffffe450   0x00007fff   0x004004df   0x00000000
              0x7fffffffe420: 0xd080fba0   0x00000032   0x00400560   0x00000000
              0x7fffffffe430: 0x00000000   0x00000000   0x004003e0   0x00000000
              0x7fffffffe440: 0xffffe530   0x00007fff   0x00000000   0x12345678
              0x7fffffffe450: 0x00000000   0x00000000   0xd081ed1d   0x00000032
```

rip	rbp	rsp
4ee	e410	e3e0

Function call stack manipulation

Before leave

```
Breakpoint 2, 400528 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe3e0

%rbp → 0x7fffffffe410

rip	rbp	rsp
528	e410	e3e0

Function call stack manipulation

Before ret

```
          40052d in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rbp

rip	rbp	rsp
52d	e450	e418

%rsp

Function call stack manipulation

After ret

```
                  4004df in main ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe420

%rbp → 0x7fffffffe450

rip	rbp	rsp
4df	e450	e420

Function call stack manipulation

Summary of 4 special move instructions:

call xyz:
  3 steps: %rsp = %rsp-8, (%rsp) = %rip+5, %rip = address(xyz)

push %rbp
  2 steps: %rsp = %rsp-8, (%rsp) = %rbp

leave
  3 steps: %rsp = %rbp, %rbp = (%rbp), %rsp = %rsp+8

ret
  2 steps: %rip = (%rsp), %rsp = %rsp+8

An aside: %rbx and %r12 through %r15 must be preserved across a function call. If they are used, they should be saved in the stack and restored before leave.

Tentative Lecture Schedule

Lecture	Date	Topic
1	10/6/2014	Introduction/Tour of Systems
2	10/8/2014	Data Representation
3	10/13/2014	Data Representation
4	10/15/2014	Machine Language
5	10/20/2014	Assembly Language
6	10/22/2014	Midterm*
7	10/27/2014	Code Optimization
8	10/29/2014	Code Optimization
9	11/3/2014	Memory Hierarchy
10	11/5/2014	Caching
11	11/10/2014	Caching
12	11/12/2014	Midterm*
13	11/17/2014	Linking
14	11/19/2014	Exception Control Flow
15	11/24/2014	Virtual Memory
16	11/26/2014	Virtual Memory
17	12/1/2014	Concurrent Programming
18	12/3/2014	Concurrent Programming
19	12/8/2014	Review
20	12/10/2014	Review
	12/16/2014	Final Exam 8AM-11AM
	11/28/2014	No Discussion - Thanksgiving Holiday
		* No office hour

Tentative Lecture Schedule

Lecture	Date	Topic
1	10/6/2014	Introduction/Tour of Systems
2	10/8/2014	Data Representation
3	10/13/2014	Data Representation
4	10/15/2014	Machine Language
5	10/20/2014	Assembly Language
6	10/22/2014	Midterm*
7	10/27/2014	Assembly Language
8	10/29/2014	Code Optimization
9	11/3/2014	Code Optimization
10	11/5/2014	Memory Hierarchy
11	11/10/2014	Caching
12	11/12/2014	Caching
13	11/17/2014	Midterm*
14	11/19/2014	Linking
15	11/24/2014	Exception Control Flow
16	11/26/2014	Virtual Memory
17	12/1/2014	Virtual Memory
18	12/3/2014	Concurrent Programming
19	12/8/2014	Concurrent Programming
20	12/10/2014	Review
	12/16/2014	Final Exam 8AM-11AM
	11/28/2014	No Discussion - Thanksgiving Holiday
		* No office hour

Assignments

Homework 2

Lab 2

Assembly Language Instructions

Arithmetic and Logical Operations

address	leal	memory,register	load effective address (address arithmetic,destination only a register)
unary	inc	register or memory	increment
	dec	register or memory	decrement
	neg	register or memory	negate
	not	register or memory	complement
arithmetic	add	memory or register,register	add
	sub	memory or register,register	subtract
	imul	memory or register,register	integer multiply
	idiv	memory or register	integer divide (divides RDX:RAX by source)
logical	xor	memory or register,register	bitwise exclusive or
	or	memory or register,register	bitwise or
	and	memory or register,register	bitwise and
shift	sal	immediate or one byte register,memory or register	left arithmetic shift (fill right with zeroes)
	shl	immediate or one byte register,memory or register	left logical shift (sal) (fill right with zeroes)
	sar	immediate or one byte register,memory or register	right arithmetic shift (fill left with sign bit)
	shr	immediate or one byte register,memory or register	right logical shift (fill left with zeroes)

only register %cl allowed for register operand

Practice Problem -- leal

assume %rax contains x, %rbx contains y

	result
leal 6(%rax), %rcx	x+6
leal (%rax,%rbx), %rcx	x+y
leal (%rax,%rbx,4), %rcx	x+4y
leal 7(%rax,%rax,8), %rcx	x+8x+7
leal 0x0a(,%rbx,4), %rcx	4y+10
leal 9(%rax,%rbx,2), %rcx	x+2y+9

Assembly Language Practice

Practice Problem -- shifts

assume %rax contains 0x800000000000000f (8 bytes) = 1000 0000 0000 0000 … 0000 0000 0000 1111 (64 bits)

	result	
sal $2,%rax	0x000000000000003c	
shl $2,%rax	0x000000000000003c	
sar $2,%rax	0xe000000000000003	
shr $2,%rax	0x2000000000000003	
sal $63,%rax	0x8000000000000000	
sar $63,%rax	0xFFFFFFFFFFFFFFFF	
leal $0x1,%rax		
sal $63,%rax		
sar $63,%rax	0xFFFFFFFFFFFFFFFF	
and 0xfffffffe,%rax		
sal $63,%rax	0x0000000000000000	// no matter whar is in %rax

using %eax allows only shifts from 0-31 positions

Practice Problem -- convert multiply to shifts

assume x is 10

x * 17	17 = 16 + 1	(x<<4) + x	10 + 160 = 170
x * -7	-7 = -4 -2 -1	-(x<<2) - (x<<1) - x	-40 - 20 - 10 = -70
	also	-(x<<3) + x	-80 + 10
x * 60	60 = 32 + 16 + 8 + 4	(x<<5) + (x<<4) + (x<<3) +(x<<2)	320 + 160 + 80 + 40 = 600
	also	(x<<6) - (x<<2)	640 - 40
x * -112	-112 = -64 – 32 – 16	-(x<<6) - (x<<5) - (x<<4)	-640 - 320 - 160 = -1120
	also	-(x<<7) + (x<<4)	-1280 + 160

What is x<<4 + x ? = x << (4 + x) = x << 14 = 163840 ! shifts have lowest precedence

Assembly Language Practice

Practice Problem -- arithmetic

assume	Address	Value	Register	Value
	0x100	0xFF	%eax	0x100
	0x104	0xAB	%ecx	0x1
	0x108	0x13	%edx	0x3
	0x10C	0x11		

Instruction	Destination	Value	
addl %ecx,(%eax)	0x100	0x101	
subl %edx,4(%eax)	0x104	0xA8	
imull $16,(%eax,%edx,4)	0x10c	0x110	0x11 = 0001 0001 * 16 = 0001 0001 0000
incl 8(%eax)	0x108	0x14	
decl %ecx	%ecx	0x0	
subl %edx,%eax	%eax	0xFD	

Status register, set after arithmetic instructions.

Status codes:

for unsigned int t, a, b and after say, t = a+b

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
(unsigned) t < (unsigned) a

ZF: Zero Flag. The most recent operation yielded zero.
t == 0

for signed int t, a, b and after t = a+b

SF: Sign Flag. The most recent operation yielded a negative value.
t < 0

OF: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.
(a<0 == b<0) && (t<0 != a<0)

Show condtest.c

Compare and test.

all sizes but must be the same. In C, comparing two different sizes, they must first be made the same (larger)size and this depends on signed/unsigned.

arithmetic	cmp	memory or register (S2), memory or register (S1)	set code depending on $S_1 - S_2$
logical	test	memory or register (S2), memory or register (S1)	set code depending on $S_{1,}\& S_2$

test %eax,%eax $S_1\&S_1 = S_1$! But status is set depending on <,=,> 0.

set saves various Boolean combinations of the condition register

Combinations of tests in one instruction.

Destination is one byte (low order byte if register)

			Condition	
-	sete	memory or register	ZF	equal / zero
-	setne	memory or register	~ZF	not equal / not zero
-	sets	memory or register	SF	negative
-	setns	memory or register	~SF	not negative
-	setg	memory or register	~(SF ^ OF) & ~ZF	greater (signed >)
-	setge	memory or register	~(SF ^ OF)	greater or equal (signed >=)
-	setl	memory or register	SF ^ OF	less (signed <)
-	setle	memory or register	~(SF ^ OF) | ZF	less or equal (signed <=)
-	seta	memory or register	~CF & ~ZF	above (unsigned >)
-	setae	memory or register	!CF	above or equal (unsigned >=)
-	setb	memory or register	CF	below (unsigned <)
-	setbe	memory or register	CF | ZF	below or equal (unsigned <=)

Status register, set after arithmetic instructions.

Status codes:

logical

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
(unsigned) t < (unsigned) a when doing t = a+b

ZF: Zero Flag. The most recent operation yielded zero.
t == 0

arithmetic

SF: Sign Flag. The most recent operation yielded a negative value.
t < 0

OF: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.
(a<0 == b<0) && (t<0 != a<0) when doing t = a+b

Show condtest.c

Compare and test.

Jump (replaces %rip) goes to the location in the instruction or memory or register (*operand) if the condition matches. Goes to the next instruction, if not.

		Synonym	Condition	
-	jmp		1	always
-	je	jz	ZF	equal / zero
-	jne	jnz	~ZF	not equal / not zero
-	js		SF	negative
-	jns		~SF	nonnegative
-	jg	jnle	~(SF ^ OF) & ~ZF	greater (signed >)
-	jge	jnl	~(SF ^ OF)	greater or equal (signed >=)
-	jl	jnge	SF ^ OF	less (signed <)
-	jle	jng	(SF ^ OF) | ZF	less or equal (signed <=)
-	ja	jnbe	~CF & ~ZF	above (unsigned >)
-	jae	jnb	~CF	above or equal (unsigned >=)
-	jb	jnae	CF	below (unsigned <)
-	jbe	jna	CF | ZF	below or equal (unsigned <=)

If Statements

IF
then
else

GoToLess Programming?

```
        if ( <condition> )
                statement if true ;
        else
                statement if false ;
```

can leave off the else

translates to in !GoToLess programming

```
        if ( !<condition> )
                goto false_part ;
true_part:
        statement if true ;
        goto do done ;
false_part:
        statement if false ;
done: ;
```

It compiles into exactly the !GoToLess way. (jmp is the assembly language equivalent of goto)

If Statements

Compilation?

```
int main()
  {
  int i = 1, j =2, a =3, b = 4 , result =0 ;

  if ( i < j )
    result = a ;
  else
    result = b ;

  return 0 ;
  }

  400478 <+4>:    movl   $0x1,-0x14(%rbp)   // i
  40047f <+11>:   movl   $0x2,-0x10(%rbp)   // j
  400486 <+18>:   movl   $0x3,-0xc(%rbp)    // a
  40048d <+25>:   movl   $0x4,-0x8(%rbp)    // b
  400494 <+32>:   movl   $0x0,-0x4(%rbp)    // result
  40049b <+39>:   mov    -0x14(%rbp),%eax   // i
  40049e <+42>:   cmp    -0x10(%rbp),%eax   // compare to j
  4004a1 <+45>:   jge    0x4004ab <main+55> // if >= goto false_part
  4004a3 <+47>:   mov    -0xc(%rbp),%eax    // true_part: a
  4004a6 <+50>:   mov    %eax,-0x4(%rbp)    // result
  4004a9 <+53>:   jmp    0x4004b1 <main+61> // goto done
  4004ab <+55>:   mov    -0x8(%rbp),%eax    // false_part: b
  4004ae <+58>:   mov    %eax,-0x4(%rbp)    // result
  4004b1 <+61>                              // done:
```

Do While Statements

Do While

GoToLess

```
do {
        statement(s) }
        while ( <condition> ) ;
```

!GoToLess

```
loop:
    statement(s)
    If ( <condition> )
        goto loop ;
```

Test is <u>after</u> statement(s) so goes through at least once.

Do While Statements

Compilation

```
int main()
  {
  int n = 20, result = 1 ;

  do {
    result = result*n ;
    n = n-1 ;
    } while ( n > 1 ) ;

  return 0 ;
  }

  400478 <+4>:    movl   $0x14,-0x8(%rbp) // n
  40047f <+11>:   movl   $0x1,-0x4(%rbp)  // result
  400486 <+18>:   mov    -0x4(%rbp),%eax  // result
  400489 <+21>:   imul   -0x8(%rbp),%eax  // times n
  40048d <+25>:   mov    %eax,-0x4(%rbp)  // result
  400490 <+28>:   subl   $0x1,-0x8(%rbp)  // minus 1
  400494 <+32>:   cmpl   $0x1,-0x8(%rbp) // <condition>
  400498 <+36>:   jg     0x400486 <main+18>
```

While Statements

While

While statement

GoToLess

```
while ( <condition> ) ; {
        statement(s) }
```

!GoToLess

```
loop:
    If (! <condition> )
        goto done ;
    statement(s)
    goto loop ;
done: ;
```

Test is <u>before</u> statement(s) so possibly goes through not at all.

While Statements

Compilation

```
int main()
  {
  int n = 20, result = 1 ;

  while ( n > 1 ) {
    result = result*n ;
    n = n-1 ;
    }

  return 0 ;
  }

  400478 <+4>:    movl   $0x14,-0x8(%rbp)
  40047f <+11>:   movl   $0x1,-0x4(%rbp)
  400486 <+18>:   jmp    0x400496 <main+34>
  400488 <+20>:   mov    -0x4(%rbp),%eax
  40048b <+23>:   imul   -0x8(%rbp),%eax
  40048f <+27>:   mov    %eax,-0x4(%rbp)
  400492 <+30>:   subl   $0x1,-0x8(%rbp)
  400496 <+34>:   cmpl   $0x1,-0x8(%rbp)
  40049a <+38>:   jg     0x400488 <main+20>
```

For Statements

For

for statement is the same as a while statement

GoToLess

```
for( init-expr; test-expr; update-expr) {
  statement(s)   }
```

can be rewritten

```
init-expr ;
while ( test-expr ) {
        statement(s)
        update-expr ;
        }
```

!GoToLess

```
        init-expr ;
loop:
        if ( !test-expr )
                goto done ;
        statements(s)
        update-expr ;
        goto loop ;
done:
```

but we will see that it is actually implemented as:

```
        init-expr ;
        goto test ;
loop:   statements(s)
        update-expr ;
test:   if ( test-expr )
                goto loop ;
```

For Statements

Compilation

```
int main()
  {
  int n = 20, result = 1, i ;

  for( i=n; i>1; i-- )
    {
    result = result*n ;
    n = n-1 ;
    }

  return 0 ;
  }

  400478 <+4>:    movl   $0x14,-0xc(%rbp)   // n
  40047f <+11>:   movl   $0x1,-0x8(%rbp)    // result
  400486 <+18>:   mov    -0xc(%rbp),%eax    // init-expr
  400489 <+21>:   mov    %eax,-0x4(%rbp)    //  i
  40048c <+24>:   jmp    0x4004a0 <main+44>
  40048e <+26>:   mov    -0x8(%rbp),%eax    // result
  400491 <+29>:   imul   -0xc(%rbp),%eax    // n
  400495 <+33>:   mov    %eax,-0x8(%rbp)    // result
  400498 <+36>:   subl   $0x1,-0xc(%rbp)    // update-expr
  40049c <+40>:   subl   $0x1,-0x4(%rbp)    // i-1
  4004a0 <+44>:   cmpl   $0x1,-0x4(%rbp)    // test-expr
  4004a4 <+48>:   jg     0x40048e <main+26>
```

Conditional Move

cmov

Added to avoid branch prediction errors. Move only if true

	Instruction	Synonym		
-	cmove	cmovz	ZF	Equal / zero
-	cmovne	cmovnz	~ZF	Not equal / not zero
-	cmovs		SF	Negative
-	cmovns		~SF	Nonnegative
-	cmovg	cmovnle	~(SF ^ OF) & ~ZF	Greater (signed >)
-	cmovge	cmovnl	~(SF ^ OF)	Greater or equal (signed >=)
-	cmovl	cmovnge	SF ^ OF	Less (signed <)
-	cmovle	cmovng	(SF ^ OF) | ZF	Less or equal (signed <=)
-	cmova	cmovnbe	~CF & ~ZF	Above (unsigned >)
-	cmovae	cmovnb	~CF	Above or equal (Unsigned >=)
-	cmovb	cmovnae	CF	Below (unsigned <)
-	cmovbe	cmovna	CF |	ZF below or equal (unsigned <=)

Carefully read pages 208-212.

Conditional Assignments

comparison expresion ? true statement : false statement

```
x < y ?  y - x : x - y
```

Two ways to do this

```
if ( x < y )
        ret = y-x ;
else
        ret = x-y ;
return ret ;
```

Or:

```
reta = y-x ;
retb = x-y ;
test = x<y ;
if (test) retb = reta ; // conditional move undoes retb = x-y ;
return retb ;
```

Conditional Assignments

Compilation

```
int absdiff( int x, int y )
  {
  return x < y ? y-x : x-y ;
  }

  400493 <+10>:   mov   -0x4(%rbp),%eax
  400496 <+13>:   cmp   -0x8(%rbp),%eax         // x:y
  400499 <+16>:   jge   0x4004a9 <absdiff+32>
  40049b <+18>:   mov   -0x4(%rbp),%eax         // true: x
  40049e <+21>:   mov   -0x8(%rbp),%edx         // y
  4004a1 <+24>:   mov   %edx,%ecx
  4004a3 <+26>:   sub   %eax,%ecx
  4004a5 <+28>:   mov   %ecx,%eax
  4004a7 <+30>:   jmp   0x4004b5 <absdiff+44>
  4004a9 <+32>:   mov   -0x8(%rbp),%eax         // false: y
  4004ac <+35>:   mov   -0x4(%rbp),%edx         // x
  4004af  <+38>:  mov   %edx,%ecx
  4004b1 <+40>:   sub   %eax,%ecx
  4004b3 <+42>:   mov   %ecx,%eax
  4004b5 <+44>:   leaveq                        // done
  4004b6 <+45>:   retq
-O
  400474 <+0>:    mov   %esi,%eax
  400476 <+2>:    sub   %edi,%eax    // x-y
  400478 <+4>:    mov   %edi,%edx
  40047a <+6>:    sub   %esi,%edx    // y-x
  40047c <+8>:    cmp   %esi,%edi    // compare x:y
  40047e <+10>:   cmovge %edx,%eax
  400481 <+13>:   retq
```

Explicit list of cases

The formal definition is:

```
switch ( <expression> )
  {
  case <constant1> :
       statments1     // executed when <expression> = <constant1>
                      // if statements end with break ; goto the end
                      // otherwise, execute the next set of statements
  case <constant2> :
       statements2   // statements can be null, in which case, the next set of statements apply
     .
     .
  default :
       default statements
  }
```

Switch

Explicit list of cases

```
int switch_eg(int x, int n) {
int result = x;

switch (n) {

case 100:
result *= 13;     // x * 13
break;

case 102:
result += 10;    // x + 10
/* Fall through */

case 103:
result += 11;
break;

case 104:
case 106:
result *= result;
break;

default:
result = 0;
}

return result;
}
```

Case by case outcome?

<100
100
101
102
103
104
105
106
>106

Explicit list of cases

```
4004f3 <+4>:    mov   %edi,-0x14(%rbp)   //  x
4004f6 <+7>:    mov   %esi,-0x18(%rbp)  //  n
4004f9 <+10>:   mov   -0x14(%rbp),%eax
4004fc <+13>:   mov   %eax,-0x4(%rbp)  //  result
4004ff <+16>:   mov   -0x18(%rbp),%eax
400502 <+19>:   sub   $0x64,%eax          // n-100
400505 <+22>:   cmp   $0x6,%eax
400508 <+25>:   ja    0x40053f <switch_eg+80> // jump above: unsigned
40050a <+27>:   mov   %eax,%eax           // ?
40050c <+29>:   mov   0x400650(,%rax,8),%rax  // 0x400650+%rax*8
400514 <+37>:   jmpq  *%rax               // indirect unconditional
400516 <+39>:   mov   -0x4(%rbp),%edx       // 100
400519 <+42>:   mov   %edx,%eax
40051b <+44>:   add   %eax,%eax
40051d <+46>:   add   %edx,%eax           // shift and add to
40051f <+48>:   shl   $0x2,%eax           // multiply by 13
400522 <+51>:   add   %edx,%eax
400524 <+53>:   mov   %eax,-0x4(%rbp)
400527 <+56>:   jmp   0x400546 <switch_eg+87> // break
400529 <+58>:   addl  $0xa,-0x4(%rbp)       // 102
40052d <+62>:   addl  $0xb,-0x4(%rbp)       // 103
400531 <+66>:   jmp   0x400546 <switch_eg+87> // break
400533 <+68>:   mov   -0x4(%rbp),%eax       // 104,106
400536 <+71>:   imul  -0x4(%rbp),%eax
40053a <+75>:   mov   %eax,-0x4(%rbp)
40053d <+78>:   jmp   0x400546 <switch_eg+87> // break
40053f <+80>:   movl  $0x0,-0x4(%rbp)      // default:
400546 <+87>:   mov   -0x4(%rbp),%eax
```

Branch Table

```
400650 <+39>   // 100
400658 <+80>   // 101
400660 <+58>   // 102
400668 <+62>   // 103
400670 <+68>   // 104
400678 <+80>   // 105
400680 <+68>   // 106
```

Procedure Calls

Example

```
int called(int a, int b, int *c, int d, int e, int f) ;
void call2() ;
main()
  {
  int a, b, c, d, e, f ;
  called( a,b,&c,d,e,f ) ;
  }

int called(int a, int b, int *c, int d, int e, int f)
  {
  char x[30] ;
  int result = a+b+*c+d+e+f ;
  x[0] = 'a' ;
  call2() ;
  return result;
  }

void call2()
  {
  int a,b,c ;

  c = a+b ;
  }
```

Procedure Calls

Assembly language

```
Dump of assembler code for function main:
  400474 <+0>:    push   %rbp
  400475 <+1>:    mov    %rsp,%rbp
  400478 <+4>:    push   %rbx
  400479 <+5>:    sub    $0x28,%rsp
  40047d <+9>:    mov    -0x14(%rbp),%edi
  400480 <+12>:   mov    -0x18(%rbp),%esi
  400483 <+15>:   mov    -0x1c(%rbp),%ecx
  400486 <+18>:   lea    -0x28(%rbp),%rdx
  40048a <+22>:   mov    -0x20(%rbp),%ebx
  40048d <+25>:   mov    -0x24(%rbp),%eax
  400490 <+28>:   mov    %edi,%r9d
  400493 <+31>:   mov    %esi,%r8d
  400496 <+34>:   mov    %ebx,%esi
  400498 <+36>:   mov    %eax,%edi
  40049a <+38>:   callq  0x4004a6 <called>
  40049f <+43>:   add    $0x28,%rsp
  4004a3 <+47>:   pop    %rbx
  4004a4 <+48>:   leaveq
  4004a5 <+49>:   retq
```

Procedure Calls

Assembly language

Dump of assembler code for function called:

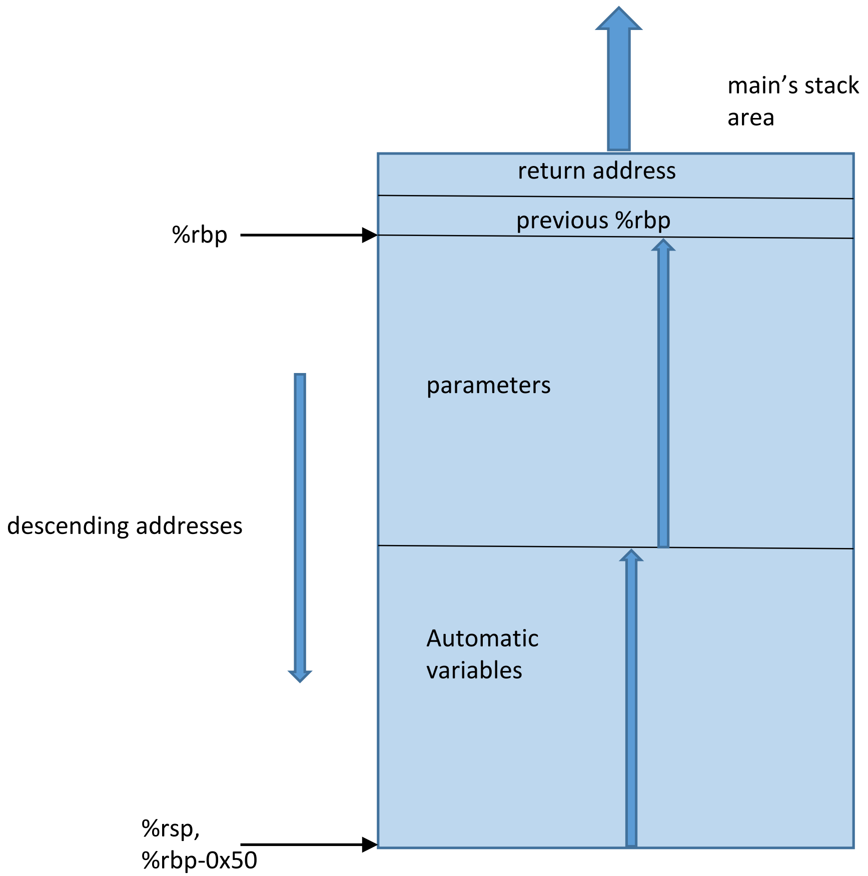
```
4004a6 <+0>:    push   %rbp
4004a7 <+1>:    mov    %rsp,%rbp
4004aa <+4>:    sub    $0x50,%rsp
4004ae <+8>:    mov    %edi,-0x34(%rbp)
4004b1 <+11>:   mov    %esi,-0x38(%rbp)
4004b4 <+14>:   mov    %rdx,-0x40(%rbp)
4004b8 <+18>:   mov    %ecx,-0x44(%rbp)
4004bb <+21>:   mov    %r8d,-0x48(%rbp)
4004bf <+25>:   mov    %r9d,-0x4c(%rbp)
4004c3 <+29>:   mov    -0x38(%rbp),%eax
4004c6 <+32>:   mov    -0x34(%rbp),%edx
4004c9 <+35>:   add    %eax,%edx
4004cb <+37>:   mov    -0x40(%rbp),%rax
4004cf <+41>:   mov    (%rax),%eax
4004d1 <+43>:   lea    (%rdx,%rax,1),%eax
4004d4 <+46>:   add    -0x44(%rbp),%eax
4004d7 <+49>:   add    -0x48(%rbp),%eax
4004da <+52>:   add    -0x4c(%rbp),%eax
4004dd <+55>:   mov    %eax,-0x4(%rbp)
4004e0 <+58>:   movb   $0x61,-0x30(%rbp)
4004e4 <+62>:   mov    $0x0,%eax
4004e9 <+67>:   callq  0x4004f3 <call2>
4004ee <+72>:   mov    -0x4(%rbp),%eax
4004f1 <+75>:   leaveq
4004f2 <+76>:   retq
```

Assembly language

```
Dump of assembler code for function call2:
  40051d <+0>:     push   %rbp
  40051e <+1>:     mov    %rsp,%rbp
  400521 <+4>:     mov    -0x8(%rbp),%eax
  400524 <+7>:     mov    -0xc(%rbp),%edx
  400527 <+10>:    lea    (%rdx,%rax,1),%eax
  40052a <+13>:    mov    %eax,-0x4(%rbp)
  40052d <+16>:    leaveq
  40052e <+17>:    retq
```

Procedure Calls

Let's have a look at the stack after we have entered "called":



Procedures which call themselves (recursive)

N! = factorial defined as $\prod_{i=1}^{n} i$ but also N! = N x (N-1)! And 1! = 1

```
#include <stdio.h>
int factorial( int n ) ;
Int kkk = 0 ;

main()
  {
  int m,n=6 ;
  m = n ;
  printf( "%d factorial is %d\n", m,factorial(n) ) ;
  return 0 ;
  }
int factorial(int n)
  {
  int result ;
  kkk = kkk+1
  if ( n<=1 )
    result = 1 ;
  else
    result = n*factorial(n-1) ;
  printf( "n= %2d kkk= %2d exit factorial result= %d\n", n,kkk,result ) ;
  return result ;
  }
```

Procedures which call themselves (recursive)

```
***build factorial machine***

kkk=  1 in factorial n=  6 result addr 0x7fff201be53c
kkk=  2 in factorial n=  5 result addr 0x7fff201be4fc
kkk=  3 in factorial n=  4 result addr 0x7fff201be4bc
kkk=  4 in factorial n=  3 result addr 0x7fff201be47c
kkk=  5 in factorial n=  2 result addr 0x7fff201be43c
kkk=  6 in factorial n=  1 result addr 0x7fff201be3fc
n=  1 kkk=  6 exit factorial result= 1
n=  2 kkk=  6 exit factorial result= 2
n=  3 kkk=  6 exit factorial result= 6
n=  4 kkk=  6 exit factorial result= 24
n=  5 kkk=  6 exit factorial result= 120
n=  6 kkk=  6 exit factorial result= 720
6 factorial is 720
```

Arrays

Arrays: they have a name and elements numbered from 0
to n-1 where n is declared

type x[20] where type is a C/C++ data type.

for char x[20], each box is 1 byte, total size is 20 bytes

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

The top row doesn't exist but is just labeling the "address" of each element.

For different types, the total size is n*(sizeof type). Remember that in X86-64
pointers are 8 bytes.

Pointer Arithmetic

In C, an array name can be treated as a pointer: *(x+i) is the value in x[i]. The address computation retains the type of x. That is x+i has the value &x[0]+i*(sizeof type of x).

&x[i] gives the address of x[i], automatically adjusted for the size of x.

*(x+i) gives the value stored at x[i] (called dereferencing)

&x[i]-x gives the value i. (not i*sizeof)

The type gives a scale to pointer arithmetic:

type	scale
void	1
char	1
short	2
int, float	4
double	8
pointer	8

```
int *p, *q ;
int x ;
p = &x ;
q = p+1 ;    // adds 4
```

The NULL pointer is 0.

Pointers can point to functions:

(int) (*fp)(int, int *) ; declares fp as a function pointer.

Arrays

Can have multiple dimensions: x[3][4] corresponds to:

	0	1	2	3
0				
1				
2				

So, the number of elements for x[n][m] is n*m. The size of the array is n*m*(sizeof type).

In linear memory. The array looks like:

0,0	0,1	0,2	0,3	1,0	1,1	1,2	1,3	2,0	2,1	2,2	2,3

So, the right most index runs the fastest.

To go from x[i][j] to x[k] $k = j * max_i + i$ and the memory address is k x sizeof(type x)

Generalizations

```
#define N 3
#define M 4

int x[N][M] ;
```

The address of x[i][j] is &x[0][0]+(i*M+j) * sizeof(int). Note that you can still address the array as *(x+I) but you have to know what you are doing.
For instance, you could

```
for( i=0; i<N; i++)
  for( j=0; j<M, j++ )
    {
    k = i*M+j ;
    l = *(x+k) ;  // compiler multiplies k by sizeof(int)
//
// is the same as
//      l = x[i][j] ;
//
    }
```

Loop compile

```
400481 40>:        push   %rbp
400482 <+1>:       mov    %rsp,%rbp
400485 <+4>:       sub    $0x60,%rsp
400489 <+8>:       movl   $0x1,-0x18(%rbp)   // i
400490 <+15>:      movl   $0x2,-0x14(%rbp)   // j
400497 <+22>:      movl   $0x3,-0x10(%rbp)   // k
40049e <+29>:      movl   $0x4,-0xc(%rbp)    // l
4004a5 <+36>:      movl   $0x63,-0x50(%rbp)  // x
4004ac <+43>:      movl   $0x62,-0x60(%rbp)  // y
4004b3 <+50>:      movl   $0x0,-0xc(%rbp)    // l
4004ba <+57>:      movl   $0x0,-0x18(%rbp)   // i
4004c1 <+64>:      jmp    0x400507 <main+134> //
4004c3 <+66>:      movl   $0x0,-0x14(%rbp)   // j
4004ca <+73>:      jmp    0x4004f9 <main+120> //
4004cc <+75>:      mov    -0x14(%rbp),%eax   // j
4004cf <+78>:      cltq
4004d1 <+80>:      mov    -0x14(%rbp),%edx   // j
4004d4 <+83>:      mov    %edx,-0x60(%rbp,%rax,4) // %rax*4
4004d8 <+87>:      mov    -0x18(%rbp),%edx   // i
4004db <+90>:      mov    -0x14(%rbp),%eax   // j
4004de <+93>:      cltq
4004e0 <+95>:      movslq %edx,%rdx          // i
4004e3 <+98>:      shl    $0x2,%rdx          // i*4
4004e7 <+102>:     add    %rax,%rdx          // i*4+j
4004ea <+105>:     mov    -0xc(%rbp),%eax    // l
4004ed <+108>:     mov    %eax,-0x50(%rbp,%rdx,4)  (i*4+j)*4
4004f1 <+112>:     addl   $0x1,-0x14(%rbp)   // j++
4004f5 <+116>:     addl   $0x1,-0xc(%rbp)    // l++
4004f9 <+120>:     cmpl   $0x3,-0x14(%rbp)   // j ... inner loop
4004fd <+124>:     jle    0x4004cc <main+75>  //
4004ff <+126>:     addl   $0x1,-0x18(%rbp)   // i++
400503 <+130>:     addl   $0x1,-0xc(%rbp)    // l++
400507 <+134>:     cmpl   $0x2,-0x18(%rbp)   // i ... outer loop
40050b <+138>:     jle    0x4004c3 <main+66>  //
```

```
#include <stdio.h>
#define N 3
#define M 4
#define TYP int

void sub1()
  {
  int i ;
  i = 100 ;
  }
int main()
  {
  TYP x[N][M] ;
  TYP y[M] ;
  int i = 1,j = 2,k = 3,l = 4,*m ;

  x[0][0] = 99 ;
  y[0] = 98 ;
  l = 0 ;
  for( i=0; i<N; i++,l++ )
    for( j=0; j<M; j++,l++ )
      {
      y[j] = j ;
      x[i][j] = l ;
      }

  for( i=0; i<N; i++ )
    for( j=0; j<M; j++ )
      {
      k = M*i+j ;
      m = *(x+k) ;
      }

  sub1() ;

  return 0 ;
  }
```

Structures

A road map of a memory area

Structures: a way of aggregating data into one place. Outcome of declaration is a new data type and creates a map of a contiguous area.

History: first widely used in COBOL (1959) (Common Business Oriented Language) supposedly self documenting. “Amazing” Grace Murray Hopper.

Fortran (1953) scientific language John Backus.

PL/I (1965) combines features of COBOL and Fortran, adds elements of LISP, very strong with structures, pointers first introduced, dynamic memory allocation. IBM

```
struct <new data type name>
  {
  <type> <var1> ;
  <type> <var2> ;
    .
    .
  } <optional name> ;
```

After declaration, <new data type name> is a data type just like char, int, etc.

Examples

```
struct movies
  {
  char title[50];
  int year;
  } ;
```

After this declaration, movies is a data type. If the <optional name> is present, it creates an instance of the data type. You can now declare:

```
movies yours, mine ;
```

Or even

```
movies favorites[50] ;
```

Memory layout for yours, mine is 50 character title followed by year. For favorites:

title[0] title[1] title[2] ... title[48] title[49] year

For favorites:

title[0][0] title[0][1] title[0][2] ... title[0][48] title[0][49] year[0]
title[1][0] title[1][1] title[1][2] ... title[1][48] title[1][49] year[1]

Data alignment

Old days: mandatory

Now: instructions work but C aligns in the interest of speed.

malloc always passes back address on 16 byte boundary

Structures

Compiled?

```
void main()
  {
  struct newtype1 {
    char a ;
    float b[10] ;
    char c ;
    int d ;
    } x ;
  struct newtype2 {
    char a ;
    double b[10] ;
    char c ;
    int d ;
    } y ;
  struct newtype3 {
    int a ;
    double b[10] ;
    short c ;
    int d ;
    }  z ;

  x.a = 0xff ;
  x.b[0] = 10  ;
  x.c = 0x44 ;
  x.d = 25 ;

  y.a = 0xfe ;
  y.b[0] = 9  ;
  y.c = 0x43 ;
  y.d = 24 ;

  z.a = 0xfd ;
  z.b[0] = 8  ;
  z.c = 0x42 ;
  z.d = 23 ;
  }
```

```
400474 <+0>:     push   %rbp
400475 <+1>:     mov    %rsp,%rbp
400478 <+4>:     sub    $0x88,%rsp
40047f <+11>:    movb   $0xff,-0x40(%rbp)    // x.a     offset -0x40
400483 <+15>:    mov    $0x4120,%eax
400488 <+20>:    mov    %eax,-0x3c(%rbp)     // x.b[0]  offset -0x3c   difference of 4
40048b <+23>:    movb   $0x44,-0x14(%rbp)    // x.c     offset -0x14  difference of 40 = 10 * 4
40048f <+27>:    movl   $0x19,-0x10(%rbp)    // x.d     offset -0x10  difference of 4

400496 <+34>:    movb   $0xfe,-0xa0(%rbp)    // y.a     offset -0xa0
40049d <+41>:    movabs $0x4022,%rax
4004a7 <+51>:    mov    %rax,-0x98(%rbp)     //  y.b[0]  offset -0x98  difference of 8
4004ae <+58>:    movb   $0x43,-0x48(%rbp)    //  y.c     offset -0x48  difference of 80 = 10 * 8
4004b2 <+62>:    movl   $0x18,-0x44(%rbp)    //  y.d     offset -0x44  difference of 4

4004b9 <+69>:    movl   $0xfd,-0x100(%rbp)   // z.a     offset -0x100
4004c3 <+79>:    movabs $0x4020,%rax
4004cd <+89>:    mov    %rax,-0xf8(%rbp)     // z.b[0]  offset -0xf8   difference of 4
4004d4 <+96>:    movw   $0x42,-0xa8(%rbp)    // z.c     offset -0xa8  difference of 80 = 10 * 8
4004dd <+105>:   movl   $0x17,-0xa4(%rbp)    // z.d     offset -0xa4   difference of 4

4004e7 <+115>:   leaveq
4004e8 <+116>:   retq
```

Many Variations

```
struct newtype2
{
  int a ;
  struct inner
    {
    float b ;
    int c[10] ;
    } y ;
  int *d ;
} x ;
```

The scope of the variable name lies inside of the structure. That is the name is not known outside of the structure unless you refer to the variable with its qualification:

```
x.a,   x.y.c[5]
```

You cannot refer to c[5] without the qualification unless char c[] exists outside of the structure. This means that you can have c both inside and outside of the structure. Confusing!

Passing as parameters

```
struct newtype1
        {
         int a ;
         float b[10] ;
         int c ;
         } x ;
```

If you want to pass a structure variable as a pointer, you can.

```
void to_sub1( struct newtype1 *x ) ;
```

then in to_sub1(struct newtype1 *x)

```
        {

        x->a = 10.5 ;
```

or

```
        (*x).a = 10.5
```

and

```
        (*x).y.b = 11.5
        }
```

but x.a no longer works.

Compiled code

```
#include <stdio.h>
  struct rect
        {
        int i;
        int j;
        struct inner
          {
          int i ;
          float l ;
          } b ;
        int c[3];
        int *p;
        } x,y ;

  void to_sub1( struct rect x ) ;
  void to_sub2( struct rect *x ) ;


int main()
  {
  int i ;

  printf( "%d\n", sizeof(x) ) ;

  i = 100 ;

  x.i   = 10 ;
  to_sub1( x ) ;
  to_sub2( &x ) ;
  return 0 ;
  }
```

```
Dump of assembler code for function main:
  4004c4 <+0>:     push   %rbp
  4004c5 <+1>:     mov    %rsp,%rbp
  4004c8 <+4>:     sub    $0x40,%rsp
  4004cc <+8>:     mov    $0x4006a8,%eax
  4004d1 <+13>:    mov    $0x28,%esi
  4004d6 <+18>:    mov    %rax,%rdi
  4004d9 <+21>:    mov    $0x0,%eax
  4004de <+26>:    callq  0x4003b8 <printf@plt>
  4004e3 <+31>:    movl   $0x64,-0x4(%rbp)
  4004ea <+38>:    movl   $0xa,0x2004cc(%rip)     # 0x6009c0 <x>
  4004f4 <+48>:    mov    0x2004c5(%rip),%rax     # 0x6009c0 <x>
  4004fb <+55>:    mov    %rax,(%rsp)
  4004ff <+59>:    mov    0x2004c2(%rip),%rax     # 0x6009c8 <x+8>
  400506 <+66>:    mov    %rax,0x8(%rsp)
  40050b <+71>:    mov    0x2004be(%rip),%rax     # 0x6009d0 <x+16>
  400512 <+78>:    mov    %rax,0x10(%rsp)
  400517 <+83>:    mov    0x2004ba(%rip),%rax     # 0x6009d8 <x+24>
  40051e <+90>:    mov    %rax,0x18(%rsp)
  400523 <+95>:    mov    0x2004b6(%rip),%rax     # 0x6009e0 <x+32>
  40052a <+102>:   mov    %rax,0x20(%rsp)
  40052f <+107>:   callq  0x400545 <to_sub1>
  400534 <+112>:   mov    $0x6009c0,%edi
  400539 <+117>:   callq  0x40056c <to_sub2>
  40053e <+122>:   mov    $0x0,%eax
  400543 <+127>:   leaveq
  400544 <+128>:   retq
```

Structures

Usage in functions

```
void to_sub1( struct rect x )
  {
  int i ;

  i = 100 ;
  x.i = 10 ;
  x.b.i =  5 ;
  i = x.j   ;
  i = x.b.i ;
  }

void to_sub2( struct rect *x )
  {
  int i ;

  i = 100 ;
  x -> i = 10 ;
  (*x).i = 10 ;

  x -> b.i = 5 ;
  (*x).b.i = 5 ;

  }
```

When passed by name, x.i and x.i.b no longer work because x is a pointer. Must use (*x). instead.

Unions

Look like structures but..

In unions, the offset is always 0. This means that each variable overlays or occupies the same storage as the other variables:

```
union u
  {
   int i ;
   unsigned char c[4] ;
   float a ;
   } examine_endian ;
```

Sound familiar? Pointers are not needed here! But it is dangerous.

Accessing examine_endian.a overwrites what is in examine_endian.i

Example

```
#include <stdio.h>

void main()
  {
  union endian
         {
         int i ;
         unsigned char c[4] ;
         float a ;
         } hw1 ;
  int j ;

  hw1.i = 19088743   ; /* should be
0x01234567 */

  printf( "\ninteger forward=  " ) ;
  for ( j=0; j<4; j++ ) /* prints c[4] */
         printf( "%02x", hw1.c[j] ) ;
  printf( "\n" ) ;

  printf( "integer backward= " ) ;
  for ( j=3; j>=0; j-- ) /* prints c[4] */
         printf( "%02x", hw1.c[j] ) ;
  printf( "\n\n" ) ;


  hw1.a = 10.01   ;

  printf( "float forward=  " ) ;
  for ( j=0; j<4; j++ ) /* prints c[4] */
         printf( "%02x", hw1.c[j] ) ;
  printf( "\n" ) ;

  printf( "float backward=  " ) ;
  for ( j=3; j>=0; j-- ) /* prints c[4] */
         printf( "%02x", hw1.c[j] ) ;
  printf( "\n\n" ) ;


  }
```

```
Dump of assembler code for function main:
 400554 <+0>:        push    %rbp
 400555 <+1>:        mov     %rsp,%rbp
 400558 <+4>:        sub     $0x10,%rsp
 40055c <+8>:        movl    $0x1234567,-0x10(%rbp)

 400603 <+175>:      mov     $0x412028f6,%eax
 400608 <+180>:      mov     %eax,-0x10(%rbp)
```

```
result of ./a.out

integer forward=   67452301
integer backward= 01234567

float forward=     f6282041
float backward=   412028f6
```

Memory Corruption

Many ways to do it

buffer overflow	gets/puts example
pointer goes wild	pointer error example
array index goes wild	

```
/* Corrupt1.c Stack corruption with gets */
#include <stdio.h>
void echo() ;

int main()
  {
  echo() ;
  printf( "%x\n", EOF ) ;
  }

void echo()
  {
  char inp[8] = "01234567890123456789O" ;

  while ( inp != NULL )
    {
    gets(inp) ;
    puts(inp) ;
    }
  }
```

Memory Corruption

Out of bounds subscript

```
/* corrupt2.c  Stack corruption with array
overflow */
#include <stdio.h>
void echo() ;
void sub2() ;

int main()
  {
  echo() ;
  printf( "%x\n", EOF ) ;
  }

void echo()
  {
  int i[2] ;
  int j ;
  int k ;

  i[0] = 4 ;
  i[1] = 3 ;
  i[2] = 2 ;
  i[3] = 1 ;
  j = i[4] ;
  i[4] = 0 ; /* destroys return address */
  i[5] = 0 ; /* destroys previous base pointer */
  *(i+4) = -1 ;
```

```
  for( k=-4; k>-20; k-- )
    i[k] = k ;

  sub2() ;
  }

void sub2()
  {
  int i = 5 ;
  int j ;

  j = i ;
  }
```

Out of bounds subscript

```
/* corrupt3.c   stack corruption storing outside of frame */
#include <stdio.h>
void echo() ;
void sub2() ;

  int main() {
  echo() ;
  printf( "%x\n", EOF ) ;
  }
void echo() {
  int i[2] ;
  int j ;
  int k ;

  for( k=-4; k>-500; k-- )
    i[k] = k ;

  sub2() ;
  }
void sub2() {
  int i = 5 ;
  int j ;

  j = i ;
  }
```

Memory Corruption

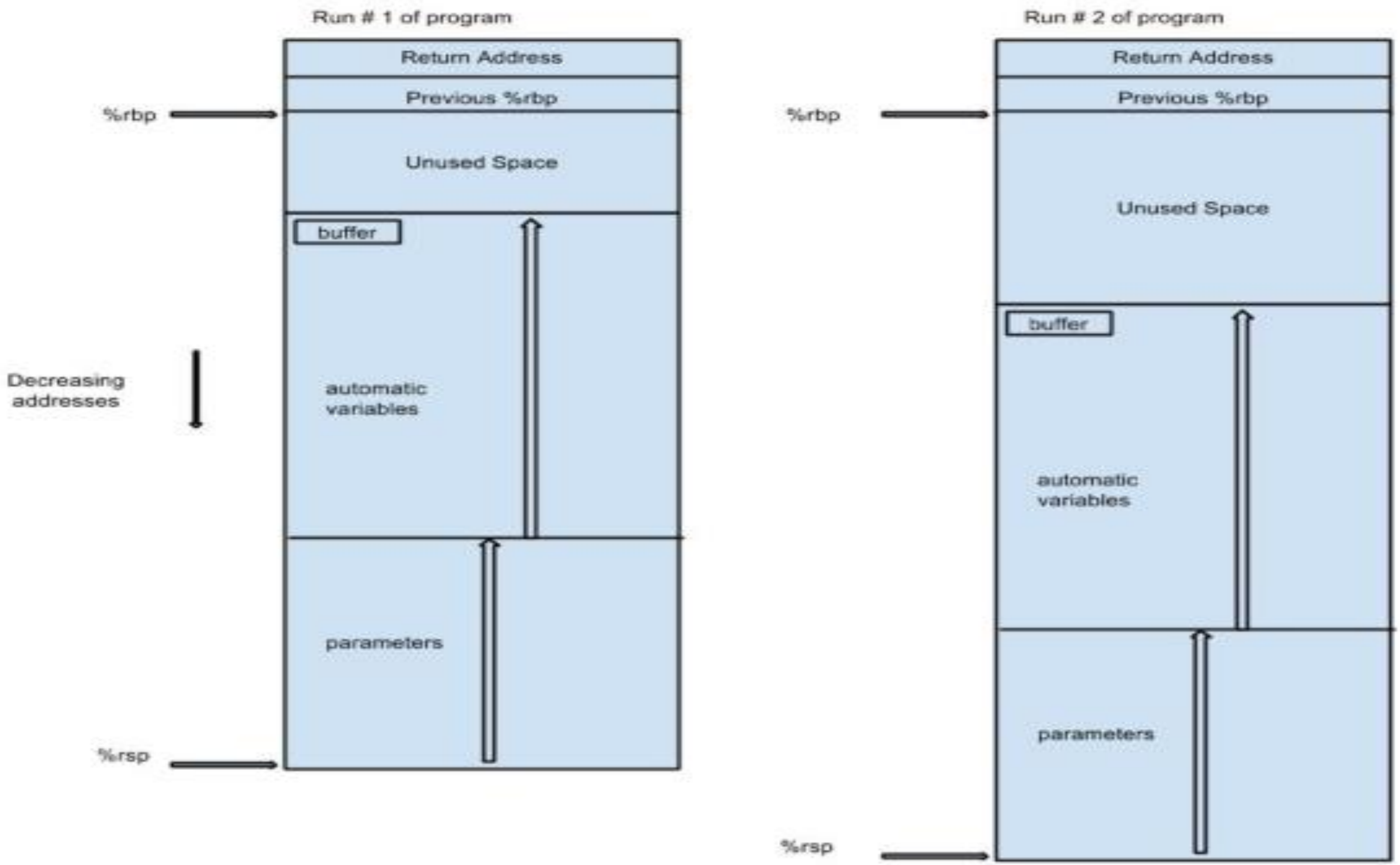
Insert code somewhere (beyond stack). Overlay the return address in the stack. When program returns, it jumps to code.

Stack randomization: after saving the return address and the previous base pointer, allocate a random amount of space in the stack. Then place the automatic variables. This way, the address of the automatic variables and the base pointer has a different offset.

Corruption detection: Store a random value somewhere in stack at the beginning of the program. Store that value in a protected area of memory. At the end of the program compare the values. If changed, raise the red flag.

Hardware which prevents pages from executing code. Memory is divided into 2K or 4K byte “pages”. Each page can be set with read/write/execute bits when in supervisory mode.

Sample Stack Randomization



Optimization

Several ways to do it

- The algorithm: smart and mindless ways
  - optimize loops
    - procedure calls
    - recomputing items which do not change
    - unrolling
    - blocking
- The optimizer
- Take advantage of architecture: parallelism, caching

Algorithms: time as a function of set size

- linear or polynomial time
- n^2 (sort1.c)
- n log n (sort2.c)
- accuracy desired? (bin packing)

Algorithms: time as a function of granularity (sort3, sort4)

The importance of measurement

- upper, lower bounds
- average behavior

Speeding up your program

Converting your program from N^2 to N logN: compare the two.

For small N it does not make much difference.

Amdahl's law.

Say T_{old} is the total time a program takes. Let's say part of the program takes a fraction f of the time. Let's speed up that part by a factor of k. then

$$T_{new} = (1-f)*T_{old}+(f*T_{old})/k = T_{old}*((1-f)+f/k)$$

then

$$T_{old} / T_{new} = 1/((1-f)+f/k) = S = \text{ speedup factor}$$

try f = 0.6, k = 3. S = 1.67

Say k is very large, then S is approximately 1/(1-f) and with f = 0.6, S = 2.5

Optimization

Compiler vs Programmer

Compiler must analyze code to see where to optimize

- reduce memory references
- take redundant code out of loops
- inline functions

Programmer can do things to allow optimization

- loop unrolling
- taking procedure calls out of loops
- reduce the use of functions: inlining
- reduce memory references
- avoid variable aliasing

Optimization

Blocks to optimization

Memory aliasing

```
  void func1(int *xp, int *yp)
  {
1    *xp += *yp;
2    *xp += *yp;
  }

  void func2(int *xp, int *yp)
  {
1    *xp += 2 * *yp;
  }

int main
  {
  int i = 10 ;

  func1( &i, &i ) ;

  i = 10 ;
  func2( &i, &i ) ;

  return 0 ;
  }
```

after line 1 in func1, i = 20, after line 2, i = 40.

after line 1 in func2, i = 30

in both functions, it "looks like" there are two distinct variables but these names are just aliases for the argument.

the compiler could try to optimize func1 to func2 and the compiled code will give different results.

Optimization

Blocks to optimization

How about:

```
x = 1000; y = 3000;
*q = y;         /* 3000 */
*p = x;         /* 1000 */
t1 = *q;        /* 1000 or 3000 */
```

if p == q, result is t1 = 1000 ; if not t1 = 3000

Can this happen with pass by value variables?

Blocks to optimization

Consider:

```
void swap(int *xp, int *yp)
  {
  *xp = *xp + *yp; /* x+y */
  *yp = *xp - *yp; /* x+y-y = x */
  *xp = *xp - *yp; /* x+y-x = y */
  }

int main()
  {
  int x,y ;

  x = 10 ;
  y = 20 ;
  swap( &x, &y ) ;

  x = 10 ;
  y = 10 ;
  swap( &x, &y ) ;

  x = 10 ;
  y = 10 ;
  swap( &x, &x ) ;

  return 0 ;
  }
```

```
Start swap
*xp  *yp
 10   20
 30   20
 30   10
 20   10
Start swap
*xp  *yp
 10   10
 20   10
 20   10
 10   10
Start swap
*xp  *yp
 10   10
 20   20
  0    0
  0    0
```

When x != y, (first two cases), everything works normally.
Even when *xp = *yp.

But when x == y, problems.

Blocks to optimization

Consider when a function operates on global variables.

```
int counter = 0;

int f()
  {
  printf( "in f() counter= %d\n", counter ) ;
  return counter++;
  }

int main()
  {
  printf( "result of f()+f()+f()+f() = %d \n", f()+f()+f()+f() ) ;

  counter = 0 ;

  printf( "result of 4*f() = %d \n", 4*f() ) ;

  return 0 ;
  }
```

```
in f() counter= 0
in f() counter= 1
in f() counter= 2
in f() counter= 3
result of f()+f()+f()+f() = 6

in f() counter= 0
result of 4*f() = 0
End value of counter = 1
```

"Optimized" program gives different results.

Measuring performance

Cycles per element

How do we measure performance? Stop watch? Program running time as a function of number of input elements. Also may be a function of the distribution of input data (coarseness).

Run the program many times with different number of input elements and data types. Plot a curve of number of elements versus running time. Fit a line to the data using least squares fit (regression) and you arrive at a formula which is

run time = constant+coefficient * N

where N is the number of input elements. So, the coefficient expresses the rate of increase in run time per additional data element. The constant expresses the overhead to start the program (run time when N = 0).

The coefficient is known as the CPE: cycles per element. Its units are run time per element. In all cases, it is relative to the speed of the computer but we think of it as cycles

Optimization

Loop unrolling

```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long int n)
{
long int i;
p[0] = a[0];
for (i = 1; i < n; i++)
  p[i] = p[i-1] + a[i];
}
```

This function computes the “prefix sum” of an array of elements: a. It is defined as:

p[0] = a[0] ;

p[i] = p[i-1]+a[i]

So, p[i] = the sum of all a[j] where j <= i

```
void psum2(float a[], float p[], long int n)
{
long int i;
p[0] = a[0];
for (i = 1; i < n-1; i+=2) {
  p[i] =  p[i-1] + a[i];
  p[i+1] = p[i] + a[i+1];
  }
/* For odd n, finish remaining element */
if (i < n)
  p[i] = p[i-1] + a[i];
  }
```

This is 1x unrolling

Loop unrolling

*Caveat** The increase in the number of lines in the code affects the savings for loop unrolling.

Lets say that it takes x units of time to execute the line of code in the loop in psum1, y units to execute the loop overhead and z time units are used when the loop is unrolled in psum2.

So, the time to execute the loop in psum1 is

$a = x*n+y*n$ execute the code plus the loop overhead

an unrolled loop program would take

$b = z*n/2+y*n/2$ execute the code half as much and the loop overhead half as much

for it to be faster, we want $b < a$ or

$z*n/2+y*n/2 < x*n+y*n$

This is the same as

$0 < x*n+y*n/2-z*n/2$

dividing by n it becomes

$0 < x+y/2-z/2 = c =$ difference in run times old - new

Loop unrolling

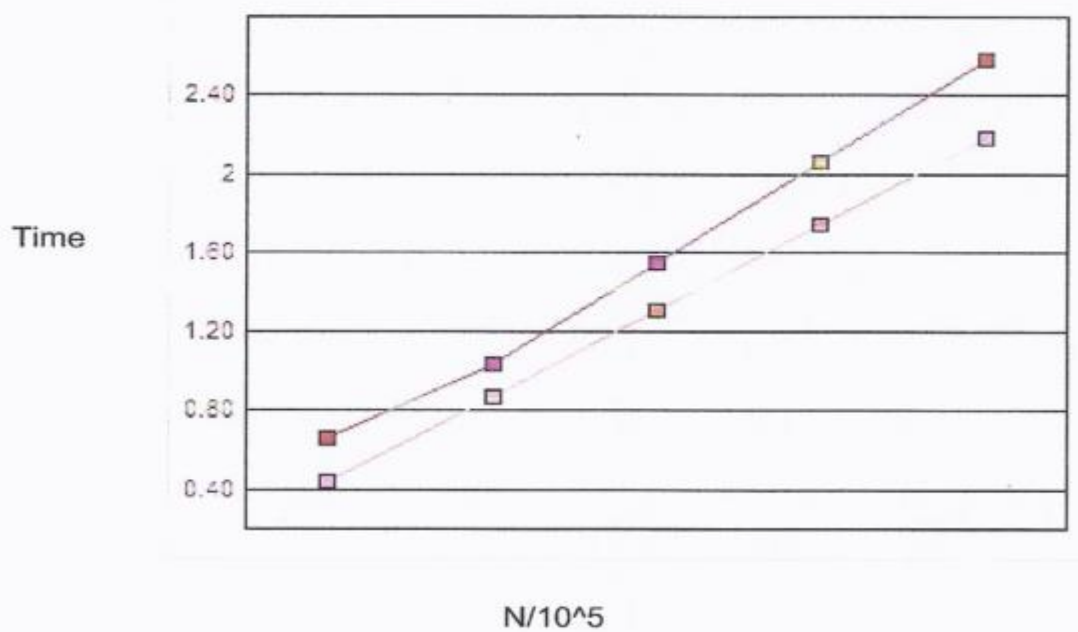
So, depending on the relative values of x, y and z, there is a diminishing rate of return!

x	y	z	c
1	1	2	0.5
1	1	3	0
1	1	4	-0.5
1	1	5	-1
2	1	3	1
2	1	4	0.5
2	1	5	0
2	1	6	-0.5

For the first line in the table, we increase the statement executions by 1 and we get a .5 improvement in the run time, by 2 and we get zero. In line 5, we increase it by 1 from 2 to 3, we get an improvement of 1.

Loop unrolling

unoptimized psum timing		psum1	psum2
	1	0.657	0.438
	2	1.034	0.870
N/10^5	3	1.550	1.311
	4	2.069	1.751
	5	2.585	2.184
constant		0.112	-0.001
coefficient		0.489	0.437



In the table, the entries are the time to run psum1(upper line) and psum2 (lower line)given the number of elements. Constant and coefficient are the terms which fit:

run time = constant + coefficient * n

Optimization

Combine example. A program to add or multiply an array of values.

```
typedef struct
  {
  int len;
  data_t *data;
  } vec_rec, *vec_ptr;
/*
* Retrieve vector element and store at dest.
* Return 0 (out of bounds) or 1 (successful)
*/
int get_vec_element(vec_ptr v, int index, data_t *dest)
  {
  if (index < 0 || index >= v->len)
  return 0;
  *dest = v->data[index];
  return 1;
  }
/* Return length of vector */
int vec_length(vec_ptr v)
  {
  return v->len;
  }
```

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine1(vec_ptr v, data_t *dest)
  {
  int i;

  *dest = IDENT;
  for (i = 0; i < vec_length(v); i++)
    {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
    }
  }
```

Disaasembling the code using GDB shows that both function calls are in the loop

Optimization

Combine example. A program to add or multiply an array of values.

Running this without optimization with 1,2,3,4,5 * 10^5
gives the following timings

combine1
Time= 1.0622 n= 100000
Time= 2.1249 n= 200000
Time= 3.1871 n= 300000
Time= 4.2490 n= 400000
Time= 5.3124 n= 500000

Running with g++ -O : optimization level 1, gives:

combine1 N
Time= 0.3437 n= 100000
Time= 0.6847 n= 200000
Time= 1.0270 n= 300000
Time= 1.3692 n= 400000
Time= 1.7116 n= 500000

A factor of 3 improvement.

Optimization

Compiled code.

```
Dump of assembler code for function _Z8combine1P7vec_recPi:
  400b1e <+0>:    push   %rbp
  400b1f <+1>:    mov    %rsp,%rbp
  400b22 <+4>:    sub    $0x20,%rsp
  400b26 <+8>:    mov    %rdi,-0x18(%rbp)
  400b2a <+12>:   mov    %rsi,-0x20(%rbp)
  400b2e <+16>:   mov    -0x20(%rbp),%rax
  400b32 <+20>:   movl   $0x0,(%rax)
  400b38 <+26>:   movl   $0x0,-0x4(%rbp)
  400b3f <+33>:   jmp    0x400b6b <+77>

  400b41 <+35>:   lea    -0x8(%rbp),%rdx
  400b45 <+39>:   mov    -0x4(%rbp),%ecx
  400b48 <+42>:   mov    -0x18(%rbp),%rax
  400b4c <+46>:   mov    %ecx,%esi
  400b4e <+48>:   mov    %rax,%rdi
  400b51 <+51>:   callq  0x400a1e <get_vec_element>
  400b56 <+56>:   mov    -0x20(%rbp),%rax
  400b5a <+60>:   mov    (%rax),%edx
  400b5c <+62>:   mov    -0x8(%rbp),%eax
  400b5f <+65>:   add    %eax,%edx
  400b61 <+67>:   mov    -0x20(%rbp),%rax
  400b65 <+71>:   mov    %edx,(%rax)
  400b67 <+73>:   addl   $0x1,-0x4(%rbp)
  400b6b <+77>:   mov    -0x18(%rbp),%rax
  400b6f <+81>:   mov    %rax,%rdi
  400b72 <+84>:   callq  0x400a69 <vec_length>
  400b77 <+89>:   cmp    -0x4(%rbp),%eax
  400b7a <+92>:   setg   %al
  400b7d <+95>:   test   %al,%al
  400b7f <+97>:   jne    0x400b41 <+35>

  400b81 <+99>:   leaveq
  400b82 <+100>:  retq
```

```
Dump of assembler code for function (with –O) _Z8combine1P7vec_recPi:
  400a7d <+0>:    movl   $0x0,(%rsi)
  400a83 <+6>:    cmpl   $0x0,(%rdi)
  400a86 <+9>:    jle    0x400ac2 <+69>
  400a88 <+11>:   push   %r12
  400a8a <+13>:   push   %rbp
  400a8b <+14>:   push   %rbx
  400a8c <+15>:   sub    $0x10,%rsp
  400a90 <+19>:   mov    %rsi,%r12
  400a93 <+22>:   mov    %rdi,%rbp
  400a96 <+25>:   mov    $0x0,%ebx

  400a9b <+30>:   lea    0xc(%rsp),%rdx
  400aa0 <+35>:   mov    %ebx,%esi
  400aa2 <+37>:   mov    %rbp,%rdi
  400aa5 <+40>:   callq  0x4009f8 <get_vec_element>
  400aaa <+45>:   mov    0xc(%rsp),%eax
  400aae <+49>:   add    %eax,(%r12)
  400ab2 <+53>:   add    $0x1,%ebx
  400ab5 <+56>:   cmp    0x0(%rbp),%ebx
  400ab8 <+59>:   jl     0x400a9b <+30>

  400aba <+61>:   add    $0x10,%rsp
  400abe <+65>:   pop    %rbx
  400abf <+66>:   pop    %rbp
  400ac0 <+67>:   pop    %r12
  400ac2 <+69>:   repz retq
```

There is no call to vec_length in sight! It has somehow figured out that the length of the vector is stored at the beginning of the vector.

Optimization

Combine example. A program to add or multiply an array of values.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine1(vec_ptr v, data_t *dest)
  {
  int i;

  *dest = IDENT;
  for (i = 0; i < vec_length(v); i++)
    {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
    }
  }
```

Disaasembling the code using GDB shows that both function calls are in the loop

Optimization

Combine example. Remove function call from loop.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine2(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);

  *dest = IDENT;
  for (i = 0; i < l                    ; i++)
    {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
    }
  }
```

Disaasembling the code using GDB shows that both function calls are in the loop

Comparison after taking a function call out of loop

Running this without optimization with 1,2,3,4,5 * 10^5
gives the following timings

combine1	combine2
Time= 1.0622 n= 100000	Time= 0.9520 n= 100000
Time= 2.1249 n= 200000	Time= 1.9029 n= 200000
Time= 3.1871 n= 300000	Time= 2.8540 n= 300000
Time= 4.2490 n= 400000	Time= 3.8053 n= 400000
Time= 5.3124 n= 500000	Time= 4.7566 n= 500000

Running with g++ -O : optimization level 1, gives:

combine1	combine2
Time= 0.3437 n= 100000	Time= 0.3054 n= 100000
Time= 0.6847 n= 200000	Time= 0.6058 n= 200000
Time= 1.0270 n= 300000	Time= 0.9087 n= 300000
Time= 1.3692 n= 400000	Time= 1.2114 n= 400000
Time= 1.7116 n= 500000	Time= 1.5146 n= 500000

A factor of 3 improvement.

Optimization

Compiled code (with –O).

with vec_length reference

```
400b1b <+30>:   lea    0xc(%rsp),%rdx
400b20 <+35>:   mov    %ebx,%esi
400b22 <+37>:   mov    %rbp,%rdi
400b25 <+40>:   callq  0x400a78 <get_vec_element>
400b2a <+45>:   mov    0xc(%rsp),%eax
400b2e <+49>:   add    %eax,(%r12)
400b32 <+53>:   add    $0x1,%ebx
400b35 <+56>:   cmp    0x0(%rbp),%ebx
400b38 <+59>:   jl     0x400b1b <+30>
```

without vec_length reference

```
400b20 <+35>:   lea    0xc(%rsp),%rdx
400b25 <+40>:   mov    %ebx,%esi
400b27 <+42>:   mov    %r12,%rdi
400b2a <+45>:   callq  0x400a78 <get_vec_element>
400b2f <+50>:   mov    0xc(%rsp),%eax
400b33 <+54>:   add    %eax,0x0(%rbp)
400b36 <+57>:   add    $0x1,%ebx
400b39 <+60>:   cmp    %r13d,%ebx
400b3c <+63>:   jne    0x400b20 <vec_recPi+35>
```

The only real difference is: no memory reference in compare!

Optimization

Horrible example of function call in a loop.

```
/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlen(const char *s)
  {
  int length = 0;
  while (*s != '\0')
    {
    s++;
    length++;
    }
  return length;
  }

/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlenfor(const char *s)
  {
  int length ;

  for( length=0;;length++ )
    if( *s == '\0' )
      break ;
    else
      s++ ;

  return length ;
  }
```

Optimization

```
/* Sample implementation of library function strlen */
/* Compute length of string */
size_t strlenif(const char *s)
  {
  int length = 0 ;

top: if( *s == '\0' )
     goto ret ;
    else
     {
     s++ ;
     length++ ;
     goto top ;
     }
ret:
  return length ;
  }
```

Optimization

```
/* Convert string to lowercase */
void lower1(char *s)
  {
  int i;

  for (i = 0; i < strlen(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
  }

/* Convert string to lowercase */
void lower2(char *s)
  {
  int i;

  for (i = 0; i < strlenfor(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
  }
```

Optimization

```
/* Convert string to lowercase */
void lower3(char *s)
  {
  int i;

  for (i = 0; i < strlenif(s); i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
  }

/* Convert string to lowercase: fast */
void lower4(char *s)
  {
  int i, n = strlenif(s) ;

  for (i = 0; i < n; i++)
    if (s[i] >= 'A' && s[i] <= 'Z')
      s[i] -= ('A' - 'a');
  }
```

Optimization

Instructive to look at timings: 4 values of N, optimized and not

Not Optimized		Optimized	
Method = While	Time= 1.0651 n= 700	Method = While	Time= 0.3778 n= 700
Method = For	Time= 1.2104 n= 700	Method = For	Time= 0.3807 n= 700
Method = If	Time= 1.2203 n= 700	Method = If	Time= 0.3799 n= 700
Method = Nofunc	Time= 0.0049 n= 700	Method = Nofunc	Time= 0.0014 n= 700
Method = While	Time= 1.3874 n= 800	Method = While	Time= 0.4919 n= 800
Method = For	Time= 1.5790 n= 800	Method = For	Time= 0.4953 n= 800
Method = If	Time= 1.6095 n= 800	Method = If	Time= 0.4944 n= 800
Method = Nofunc	Time= 0.0059 n= 800	Method = Nofunc	Time= 0.0019 n= 800
Method = While	Time= 1.7541 n= 900	Method = While	Time= 0.6214 n= 900
Method = For	Time= 1.9961 n= 900	Method = For	Time= 0.6250 n= 900
Method = If	Time= 1.9583 n= 900	Method = If	Time= 0.6240 n= 900
Method = Nofunc	Time= 0.0067 n= 900	Method = Nofunc	Time= 0.0017 n= 900
Method = While	Time= 2.1869 n= 1000	Method = While	Time= 0.7655 n= 1000
Method = For	Time= 2.4659 n= 1000	Method = For	Time= 0.7698 n= 1000
Method = If	Time= 2.3654 n= 1000	Method = If	Time= 0.7691 n= 1000
Method = Nofunc	Time= 0.0074 n= 1000	Method = Nofunc	Time= 0.0021 n= 1000

Compiled

Strlen (while)

```
4008f8 <+0>:    push   %rbp
4008f9 <+1>:    mov    %rsp,%rbp
4008fc <+4>:    mov    %rdi,-0x18(%rbp)
400900 <+8>:    movl   $0x0,-0x4(%rbp)
400907 <+15>:   jmp    0x400912 <+26>

400909 <+17>:   addq   $0x1,-0x18(%rbp)
40090e <+22>:   addl   $0x1,-0x4(%rbp)
400912 <+26>:   mov    -0x18(%rbp),%rax
400916 <+30>:   movzbl (%rax),%eax
400919 <+33>:   test   %al,%al
40091b <+35>:   jne    0x400909 <+17>

40091d <+37>:   mov    -0x4(%rbp),%eax
400920 <+40>:   cltq
400922 <+42>:   pop    %rbp
400923 <+43>:   retq
```

Strlen (for)

```
400924 <+0>:    push   %rbp
400925 <+1>:    mov    %rsp,%rbp
400928 <+4>:    mov    %rdi,-0x18(%rbp)
40092c <+8>:    movl   $0x0,-0x4(%rbp)

400933 <+15>:   mov    -0x18(%rbp),%rax
400937 <+19>:   movzbl (%rax),%eax
40093a <+22>:   test   %al,%al
40093c <+24>:   jne    0x400940 <+28>
40093e <+26>:   jmp    0x40094b <+39>
400940 <+28>:   addq   $0x1,-0x18(%rbp)
400945 <+33>:   addl   $0x1,-0x4(%rbp)
400949 <+37>:   jmp    0x400933 <+15>

40094b <+39>:   mov    -0x4(%rbp),%eax
40094e <+42>:   cltq
400950 <+44>:   pop    %rbp
400951 <+45>:   retq
```

Combine example.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine2(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);

  *dest = IDENT;
  for (i = 0; i < l                    ; i++)
    {
    data_t val;
    get_vec_element(v, i, &val);
    *dest = *dest OP val;
    }
  }
```

Combine example. Remove additional function call from loop.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine3(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  *dest = IDENT;
  for (i = 0; i < l                    ; i++)
    *dest = *dest OP data[i];
  }
```

Optimization

Comparison after taking an additional function call out of loop

Running this without optimization with 1,2,3,4,5 * 10^5
gives the following timings

combine1	combine2	Combine3
Time= 1.0622 n= 100000	Time= 0.9520 n= 100000	Time= 0.3845 n= 100000
Time= 2.1249 n= 200000	Time= 1.9029 n= 200000	Time= 0.7586 n= 200000
Time= 3.1871 n= 300000	Time= 2.8540 n= 300000	Time= 1.1377 n= 300000
Time= 4.2490 n= 400000	Time= 3.8053 n= 400000	Time= 1.5170 n= 400000
Time= 5.3124 n= 500000	Time= 4.7566 n= 500000	Time= 1.8961 n= 500000

Running with g++ -O : optimization level 1, gives:

combine1	combine2	Combine3
Time= 0.3437 n= 100000	Time= 0.3054 n= 100000	Time= 0.2312 n= 100000
Time= 0.6847 n= 200000	Time= 0.6058 n= 200000	Time= 0.4536 n= 200000
Time= 1.0270 n= 300000	Time= 0.9087 n= 300000	Time= 0.6811 n= 300000
Time= 1.3692 n= 400000	Time= 1.2114 n= 400000	Time= 0.9095 n= 400000
Time= 1.7116 n= 500000	Time= 1.5146 n= 500000	Time= 1.1332 n= 500000

Optimization

Compiled code (with –O).

without vec_length reference

```
400afd <+0>:    push   %r13
400aff <+2>:    push   %r12
400b01 <+4>:    push   %rbp
400b02 <+5>:    push   %rbx
400b03 <+6>:    sub    $0x10,%rsp
400b07 <+10>:   mov    (%rdi),%r13d
400b0a <+13>:   movl   $0x0,(%rsi)
400b10 <+19>:   test   %r13d,%r13d
400b13 <+22>:   jle    0x400b3e <+65>
400b15 <+24>:   mov    %rsi,%rbp
400b18 <+27>:   mov    %rdi,%r12
400b1b <+30>:   mov    $0x0,%ebx

400b20 <+35>:   lea    0xc(%rsp),%rdx
400b25 <+40>:   mov    %ebx,%esi
400b27 <+42>:   mov    %r12,%rdi
400b2a <+45>:   callq  0x400a78 <_Z15get_vec_elementP7vec_reciPi>
400b2f <+50>:   mov    0xc(%rsp),%eax
400b33 <+54>:   add    %eax,0x0(%rbp)
400b36 <+57>:   add    $0x1,%ebx
400b39 <+60>:   cmp    %r13d,%ebx
400b3c <+63>:   jne    0x400b20 <+35>

400b3e <+65>:   add    $0x10,%rsp
400b42 <+69>:   pop    %rbx
400b43 <+70>:   pop    %rbp
400b44 <+71>:   pop    %r12
400b46 <+73>:   pop    %r13
400b48 <+75>:   retq
```

without get_vec_start reference in loop

```
400afd <+0>:    mov    (%rdi),%edx
400aff <+2>:    mov    0x8(%rdi),%rcx
400b03 <+6>:    movl   $0x0,(%rsi)
400b09 <+12>:   test   %edx,%edx
400b0b <+14>:   jle    0x400b25 <+40>
400b0d <+16>:   mov    %rcx,%rax
400b10 <+19>:   sub    $0x1,%edx
400b13 <+22>:   lea    0x4(%rcx,%rdx,4),%rcx

400b18 <+27>:   mov    (%rax),%edx
400b1a <+29>:   add    %edx,(%rsi)
400b1c <+31>:   add    $0x4,%rax
400b20 <+35>:   cmp    %rcx,%rax
400b23 <+38>:   jne    0x400b18 <+27>

400b25 <+40>:   repz retq
```

Two memory references in loop

Also eliminates stack and argument preparation

Combine example.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine3(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  *dest = IDENT;

 for (i = 0; i < l                    ; i++)
    *dest = *dest OP data[i];
  }
```

Optimization

Combine example, eliminate additional memory reference.

data_t is a data type: int or float or double.
IDENT is 0 or 1
OP is * or +

```
void combine4(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  data_t  acc = IDENT ;

  for (i = 0; i < l                   ; i++)
   acc = acc OP data[i];

  *dest = acc ;
  }
```

Optimization

Comparison after eliminating additional memory reference

Running this without optimization with 1,2,3,4,5 * 10^5
gives the following timings

combine1	combine2	combine3	combine4
Time= 1.0622 n= 100000	Time= 0.9520 n= 100000	Time= 0.3845 n= 100000	Time= 0.3614 n= 100000
Time= 2.1249 n= 200000	Time= 1.9029 n= 200000	Time= 0.7586 n= 200000	Time= 0.7228 n= 200000
Time= 3.1871 n= 300000	Time= 2.8540 n= 300000	Time= 1.1377 n= 300000	Time= 1.0837 n= 300000
Time= 4.2490 n= 400000	Time= 3.8053 n= 400000	Time= 1.5170 n= 400000	Time= 1.4443 n= 400000
Time= 5.3124 n= 500000	Time= 4.7566 n= 500000	Time= 1.8961 n= 500000	Time= 1.8057 n= 500000

Running with g++ -O : optimization level 1, gives:

combine1	combine2	combine3	combine4
Time= 0.3437 n= 100000	Time= 0.3054 n= 100000	Time= 0.2312 n= 100000	Time= 0.0784 n= 100000
Time= 0.6847 n= 200000	Time= 0.6058 n= 200000	Time= 0.4536 n= 200000	Time= 0.1515 n= 200000
Time= 1.0270 n= 300000	Time= 0.9087 n= 300000	Time= 0.6811 n= 300000	Time= 0.2270 n= 300000
Time= 1.3692 n= 400000	Time= 1.2114 n= 400000	Time= 0.9095 n= 400000	Time= 0.3026 n= 400000
Time= 1.7116 n= 500000	Time= 1.5146 n= 500000	Time= 1.1332 n= 500000	Time= 0.3786 n= 500000

Optimization

Compiled code (with –O).

without get_vec_start reference in loop

```
400afd <+0>:   mov   (%rdi),%edx
400aff <+2>:   mov   0x8(%rdi),%rcx
400b03 <+6>:   movl  $0x0,(%rsi)
400b09 <+12>:  test  %edx,%edx
400b0b <+14>:  jle   0x400b25 <+40>
400b0d <+16>:  mov   %rcx,%rax
400b10 <+19>:  sub   $0x1,%edx
400b13 <+22>:  lea   0x4(%rcx,%rdx,4),%rcx

400b18 <+27>:  mov   (%rax),%edx
400b1a <+29>:  add   %edx,(%rsi)
400b1c <+31>:  add   $0x4,%rax
400b20 <+35>:  cmp   %rcx,%rax
400b23 <+38>:  jne   0x400b18 <+27>

400b25 <+40>:  repz retq
```

allowing local accumulator

```
400afd <+0>:   mov   (%rdi),%edx
400aff <+2>:   mov   0x8(%rdi),%rcx
400b03 <+6>:   test  %edx,%edx
400b05 <+8>:   jle   0x400b24 <+39>
400b07 <+10>:  mov   %rcx,%rax
400b0a <+13>:  sub   $0x1,%edx
400b0d <+16>:  lea   0x4(%rcx,%rdx,4),%rcx
400b12 <+21>:  mov   $0x0,%edx

400b17 <+26>:  add   (%rax),%edx
400b19 <+28>:  add   $0x4,%rax
400b1d <+32>:  cmp   %rcx,%rax
400b20 <+35>:  jne   0x400b17 <+26>

400b22 <+37>:  jmp   0x400b29 <+44>
400b24 <+39>:  mov   $0x0,%edx
400b29 <+44>:  mov   %edx,(%rsi)
400b2b <+46>:  retq
```

One memory references in loop, one less instruction.

Summary of combine improvements

remove function call whose value does not change from loop:	combine2
access data directly rather than through function call:	combine3
reduce memory references by using local variable:	combine4

These changes, by themselves can decrease the running time. But also, they assist the compiler in optimizing the program.

We ran each version using 5 different size of input data: 100000, 200000, 300000, 400000 and 500000 elements in the array to add up.

Using regression, we find the CPE (relative values) for each version:

	1	2	3	4
Unoptimized	1.06	0.95	0.38	0.36
Optmized	0.34	0.31	0.23	0.08

Optimization

The horizontal lines demarcate the for loop. The rest of the code is just overhead and we arte mainly interested in the loop.

		memory	arith	reg	func
<0>: push %r12	stack	1			
<2>: push %rbp	"	1			
<3>: push %rbx	"	1			
<4>: sub $0x10,%rsp	"		1	1	
<8>: mov %rdi,%rbp	v			1	
<11>: mov %rsi,%r12	dest			1	
<14>: movl $0x0,(%rsi)	*dest = 0 ;	1			
<20>: cmpl $0x0,(%rdi)	comp 0 to v.len	1	1		
<23>: jle <61>	if <= go to finish				
<25>: mov $0x0,%ebx	i = 0			1	
<30>:--->lea 0xc(%rsp),%rdx	parameter &val		1	1	
<35>: mov %ebx,%esi	parameter i			1	
<37>: mov %rbp,%rdi	parameter v			1	
<40>: callq <get_vec_element>					1
<45>: mov 0xc(%rsp),%eax	val	1			
<49>: add %eax,(%r12)	+ *dest	1	1		
<53>: add $0x1,%ebx	i = i+1		1		
<56>: cmp 0x0(%rbp),%ebx	compare to v.len	1	1		
<59>:--->jl <30>	if less go to loop			1	
<61>: add $0x10,%rsp	stack manipulation		1		
<65>: pop %rbx	“	1			
<66>: pop %rbp	“	1			
<67>: pop %r12	“	1			
<69>: retq	“				
	inside loop	3	4	4	1

Formal model of the loop

Let’s say that a memory operation is M cycles, an arithemtic operation is A cycles. a register operation is R cycles and a function is F cycles. Lets add 1 cycle for each byte of instruction. The length of the code is 31 bytes.

So, our loop will take M*3+A*4+R*4+F+31 cycles.

If we guess M=7, A=5, R=1 and F=15, the loop will take 91 cycles.

Removing function call: combine2

```
combine1
<30>:--->lea   0xc(%rsp),%rdx
<35>:   mov   %ebx,%esi
<37>:   mov   %rbp,%rdi
<40>:   callq <get_vec_element>
<45>:   mov   0xc(%rsp),%eax
<49>:   add   %eax,(%r12)
<53>:   add   $0x1,%ebx
<56>:   cmp   0x0(%rbp),%ebx
<59>:--->jl   <30>
```

```
combine2
<35>:--->lea   0xc(%rsp),%rdx
<40>:   mov   %ebx,%esi
<42>:   mov   %r13,%rdi
<45>:   callq <_get_vec_element>
<50>:   mov   0xc(%rsp),%eax
<54>:   add   %eax,0x0(%rbp)
<57>:   add   $0x1,%ebx
<60>:   cmp   %r12d,%ebx
<63>:--->jne  <35>
```

But, we see that combine1 has

add %eax,(%r12)   combine2 has add %eax,0x0(%rbp) : one less memory op.

Also combine1 has

cmp 0x0(%rbp),%ebx combine2 has cmp %r12d,%ebx

combine2 is one byte shorter. Significant? 3.3% = (31/30)*100

M*2+A*4+R*4+F+30 cycles = 83

CPE improvement: .31/.34 = .912   Cycles improvement = 83/91 = .911

Optimization

Removing get_vec_element function call: combine3

Combine3

	memory	arith	reg	func
<0>: mov (%rdi),%ecx				
<2>: mov 0x8(%rdi),%rdi				
<6>: movl $0x0,(%rsi)				
<12>: test %ecx,%ecx				
<14>: jle <34>				
<16>: mov $0x0,%eax				
<21>:--->mov (%rdi,%rax,4),%edx	1			
<24>: add %edx,(%rsi)	1	1		
<26>: add $0x1,%rax		1		
<30>: cmp %eax,%ecx		1		
<32>:--->jg <21>			1	
<34>: repz retq				
inside loop	2	3	1	0

Loop code is 13 bytes

M*2+A*3+R*1+0*F+13 cycles = 43

CPE improvement: .23/.31 = ..742   Cycles improvement = 46/83 = .518 not as good

adding local accumulator: combine4

Combine4

	memory	arith	reg	func
<0>: mov (%rdi),%ecx				
<2>: mov 0x8(%rdi),%rdi				
<6>: test %ecx,%ecx				
<8>: jle <i+33>				
<10>: mov $0x0,%eax				
<15>: mov $0x0,%edx				
<20>:--->add (%rdi,%rax,4),%edx	1	1		
<23>: add $0x1,%rax		1		
<27>: cmp %eax,%ecx		1		
<29>:--->jg <20>			1	
<31>: jmp 0x400aa4 <i+38>				
<33>: mov $0x0,%edx				
<38>: mov %edx,(%rsi)				
<40>: retq				
inside loop	1	3	1	0

Here we have one less memory reference and one less instruction.

M*1+A*3+R*1+0*F+11 cycles = 34

CPE improvement: .08/.23 = ..348   Cycles improvement = 34/46 = .739 not as good

conclusion

Our estimates of M, R, A and F aren’t very good.

Suggestion:

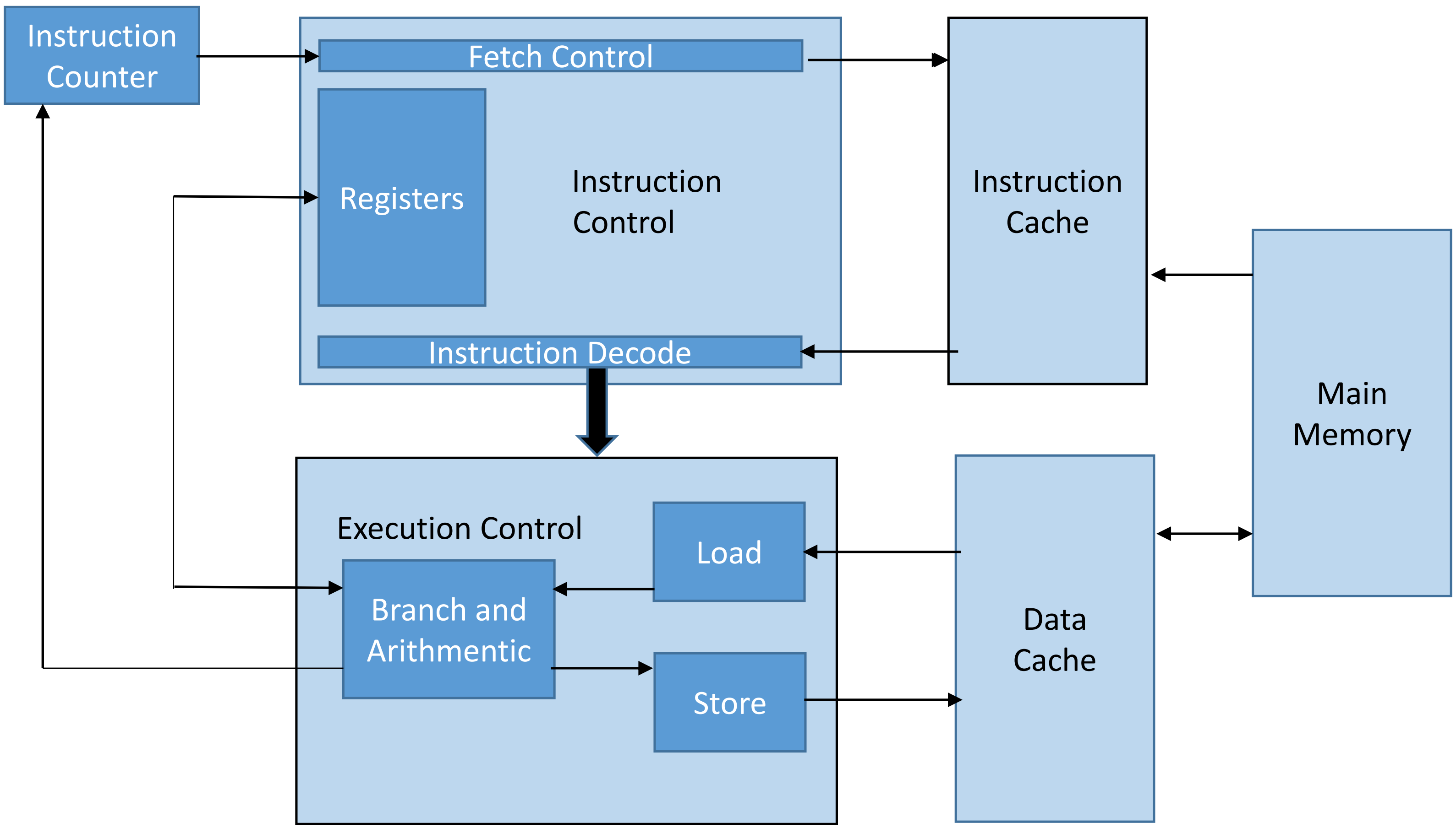
Our model is: a*M + b*R + c*A + d*F + e*S

Where M is the number of cycles for a memory operation
R is the number of cycles for a register operation
A is the number of cycles for a arithmetic operation
F is the number of cycles for a function.
S is the number of cycles penalty per byte for instruction fetch

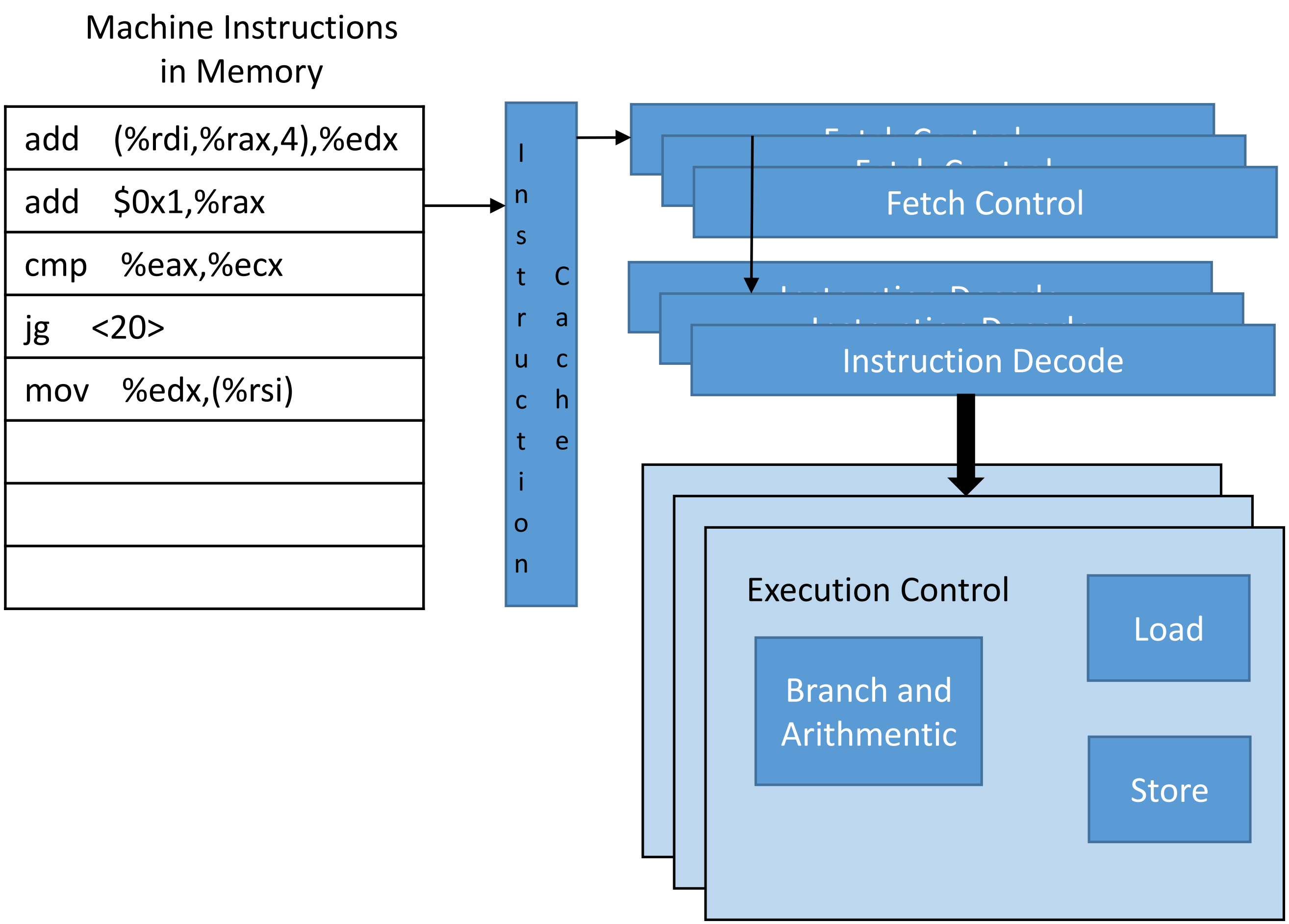
The a, b, c, d are the number of occurrences of each type of operation and s is the size of the code in bytes for a particular loop.

Run many different instances of loops, gather the CPEs, perform **multiple** regression to solve for the times of each type. But we need a way to compensate for overhead.

Machine organization



Steps in CPU operation



Optimization

instruction breakdown

Fixed point add

add   %ecx,0x0c(%edx)

load 0x0c(%edx)

add %exc

store 0x0c(%edx)

Floating point multiply

mulss x0c(%edx),%xmm0

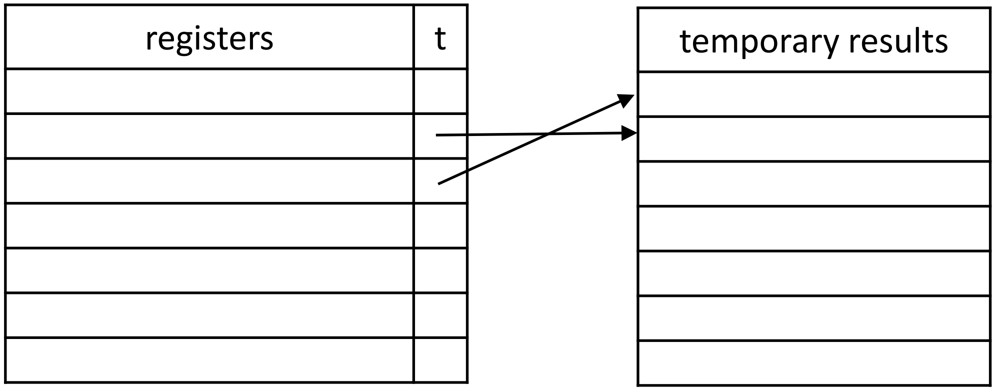
load 0x0c(%edx)

multiply %exc
  work with exponents
  work with mantissas
  round off

replace %xmm0

register renaming or aliasing

During pipelined instructions:

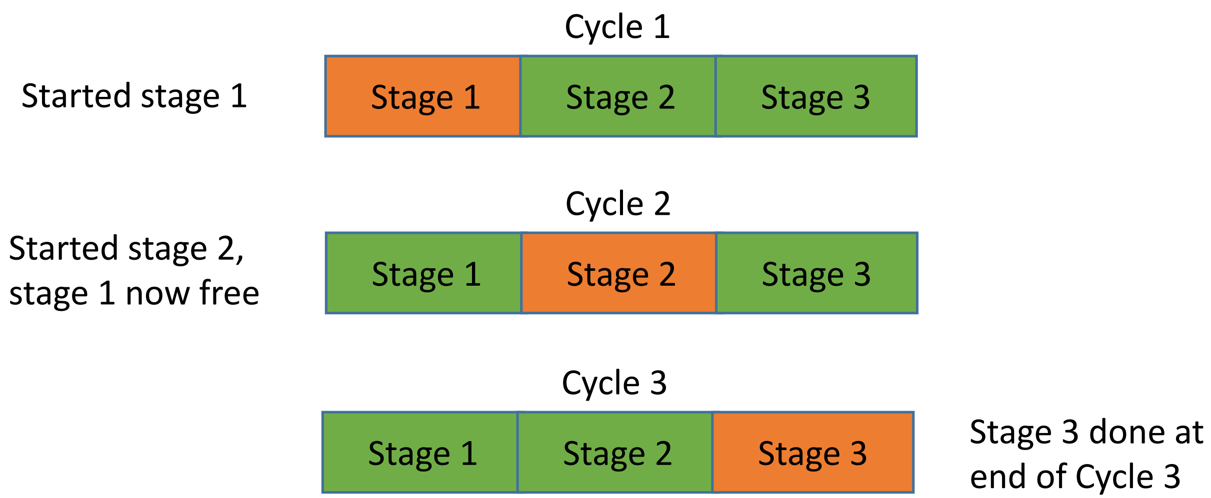


latency, issue time, throughput

Latency and issue time is measured in cycles.

latency: cycles from start to finish: how fast can you do it, once you have started.

issue time: minimum number of cycles between successive operations. How long do you have to wait before you can do it again? Can be due to multiple units or the operation is done in stages.



Latency: 3, issue time: 1

Throughput is 1/(issue time) = number of unrelated operations possible in one cycle.

Intel I7 measured latencies, issue times

Division is very high, data dependent. Too complicated for our discussions. Focus on add, multiply for integer and floating point.

All issue times for those are 1.

If the issue time is 1, you can do 1 instruction per cycle, when the latency is 1. If the latency is > 1, the instructions must be "unrelated" if the next operation needs the outcome of the prior operation in order to do 1 instruction per cycle.

	I+	I*	F+	F*	D*
Latency	1	3	3	4	5
Throughput	1	1	1	1	1

I = Integer, F = Floating point single, D = Floating point double,
+ = add, * = multiply

Latency gives the maximum time, if you have to wait for the previous result.

Throughput gives the minimum time if you can execute them "maximum pipelined".

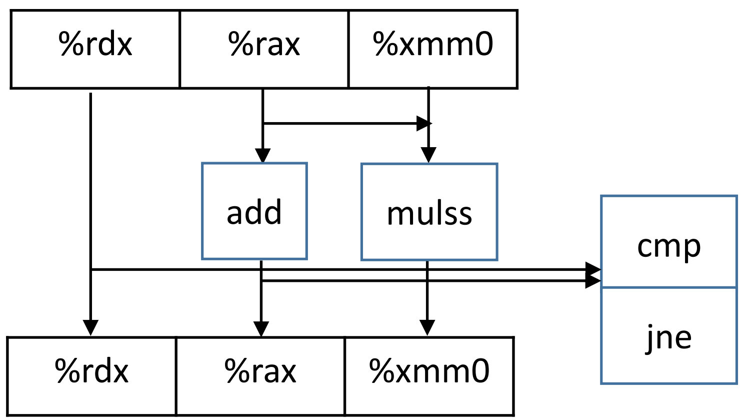
The trick is to determine whether we are latency or throughput bound

Consider the following loop (combine4 with OP = *, IDENT = 1)

```
mulss  (%rax),%xmm0
add    $0x4,%rax
cmp    %rdx,%rax
jne    loop
```

There are 3 registers involved: how do they change and how are they involved in the loop?

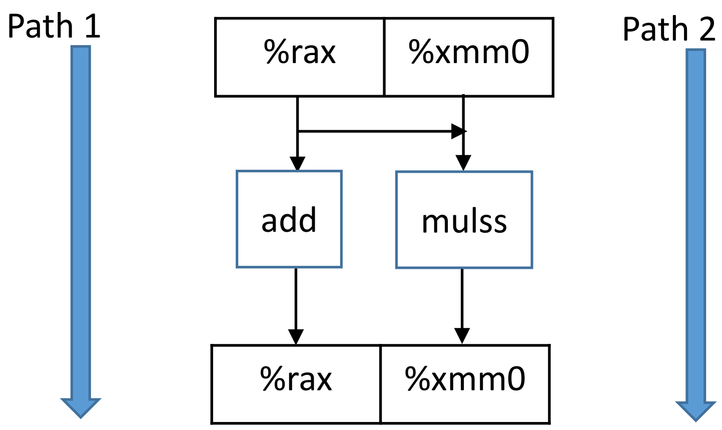
Read-only, write-only, local, loop.



Which operations depend on the outcome of the previous iteration of the loop?

Eliminate the dependencies which are not loop dependent

```
mulss  (%rax),%xmm0
add    $0x4,%rax
cmp    %rdx,%rax
jne    loop
```



Two paths: which path has the highest latency? No-brainer: add path: latency 1, mullss path: latency 4. It is the “critical” path which determines the CPE for the combine function.

The CPE is not additive but you have to figure out the overlapping operations.

Back to loop unrolling

data_t is a data type: int or float or double.
IDENT is 1
OP is *

```
void combine4(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  data_t  acc = IDENT ;

  for (i = 0; i < l     ; i++)
     acc = acc OP data[i];

  *dest = acc ;
  }
```

Optimization

Half the number of times, do the OP twice

data_t is a data type: int or float or double.
IDENT is 1
OP is *

```
void combine5(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  data_t  acc = IDENT ;

  for (i = 0; i < l     ; i+=2)
     acc = ( acc OP data[i] ) OP data[i+1];

  for(  ;   i<l ; i++ )
     acc = acc OP data[i] ;

  *dest = acc ;
  }
```

Timings for multiplication (OP = *, IDENT = 1, data_t = float)

Combine4	Combine5 (unroll x 2)
Time= 0.1527 n= 100000	Time= 0.1515 n= 100000
Time= 0.3016 n= 200000	Time= 0.3016 n= 200000
Time= 0.4523 n= 300000	Time= 0.4523 n= 300000
Time= 0.6029 n= 400000	Time= 0.6029 n= 400000
Time= 0.7537 n= 500000	Time= 0.7536 n= 500000

Expected twice the improvement but got none! Why?

Compiled code

Combine4

```
400b28 <+29>:   mulss  (%rax),%xmm0
400b2c <+33>:   add    $0x4,%rax
400b30 <+37>:   cmp    %rdx,%rax
400b33 <+40>:   jne    0x400b28 <+29>
```

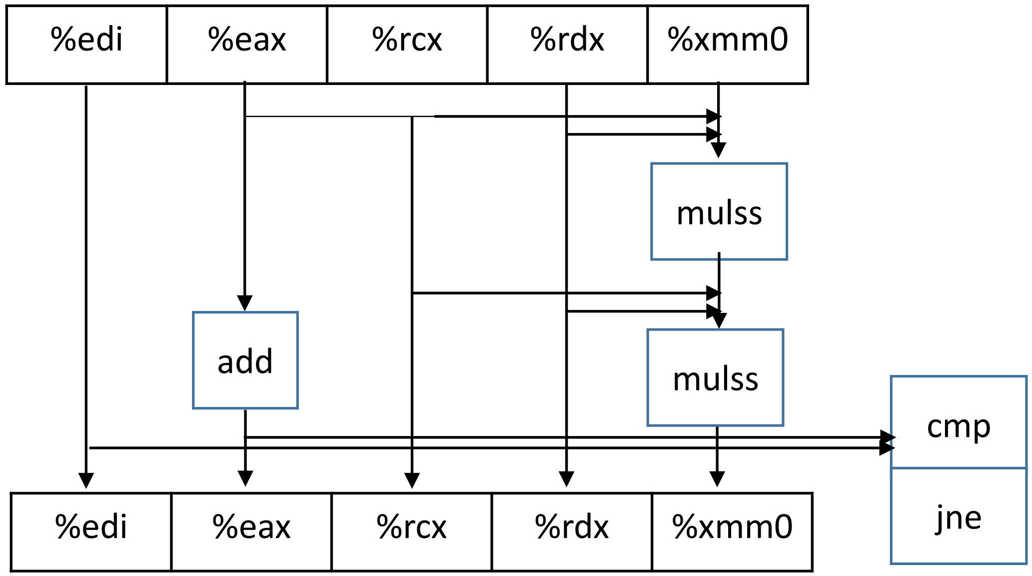
Combine5

```
400b27 <+28>:   movslq %eax,%rcx
400b2a <+31>:   mulss  (%rdx,%rcx,4),%xmm0
400b2f <+36>:   mulss  0x4(%rdx,%rcx,4),%xmm0
400b35 <+42>:   add    $0x2,%eax
400b38 <+45>:   cmp    %edi,%eax
400b3a <+47>:   jl     0x400b27 <+28>
```

Flow diagram

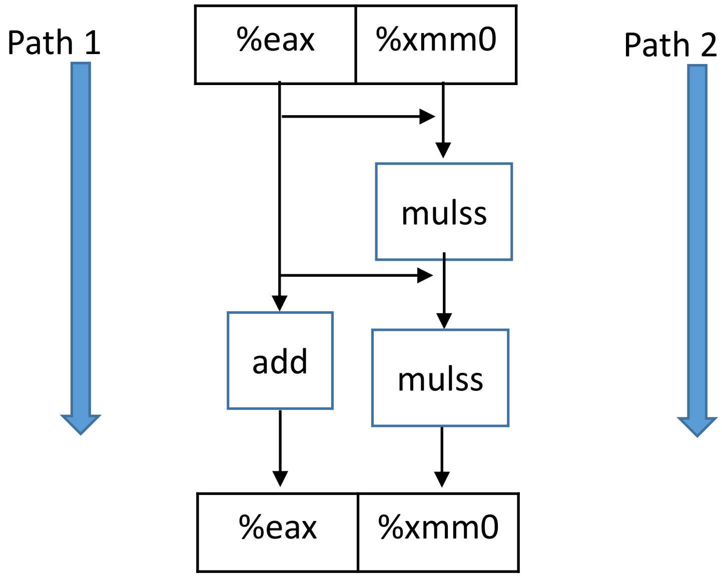
combine5 with OP = *, IDENT = 1

```
movslq %eax,%rcx
mulss  (%rdx,%rcx,4),%xmm0
mulss  0x4(%rdx,%rcx,4),%xmm0
add    $0x2,%eax
cmp    %edi,%eax
 jl    0x400b27 <+28>
```



Which operations depend on the outcome of the previous iteration of the loop?

Eliminate read only and local registers



Path 2 is the critical path with latency of 8: done half of the time

Increase parallelism by using two accumulators

data_t is a data type: int or float or double.
IDENT is 1
OP is *

```
void combine6(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  data_t  acc1 = IDENT ;
  data_t  acc2 = IDENT ;

  for (i = 0; i < l     ; i+=2)
     {
     acc1 = acc1 OP data[i]  ;
     acc2 = acc2 OP data[i+1]  ;
     }

  for(  ;   i<l ; i++ )
     acc1 = acc1 OP data[i] ;

  *dest = acc1 OP acc2 ;
  }
```

Timings for multiplication (OP = *, IDENT = 1, data_t = float)

Combine5 (unroll x 2)
Time= 0.1515 n= 100000
Time= 0.3016 n= 200000
Time= 0.4523 n= 300000
Time= 0.6029 n= 400000
Time= 0.7536 n= 500000

Combine6(unroll x 2), multiple accumulators
Time= 0.0783 n= 100000
Time= 0.1511 n= 200000
Time= 0.2264 n= 300000
Time= 0.3019 n= 400000
Time= 0.3775 n= 500000

Success! Why?

Compiled code

Combine5

```
400b27 <+28>:   movslq %eax,%rcx
400b2a <+31>:   mulss  (%rdx,%rcx,4),%xmm0
400b2f <+36>:   mulss  0x4(%rdx,%rcx,4),%xmm0
400b35 <+42>:   add    $0x2,%eax
400b38 <+45>:   cmp    %edi,%eax
400b3a <+47>:   jl     0x400b27 <+28>
```

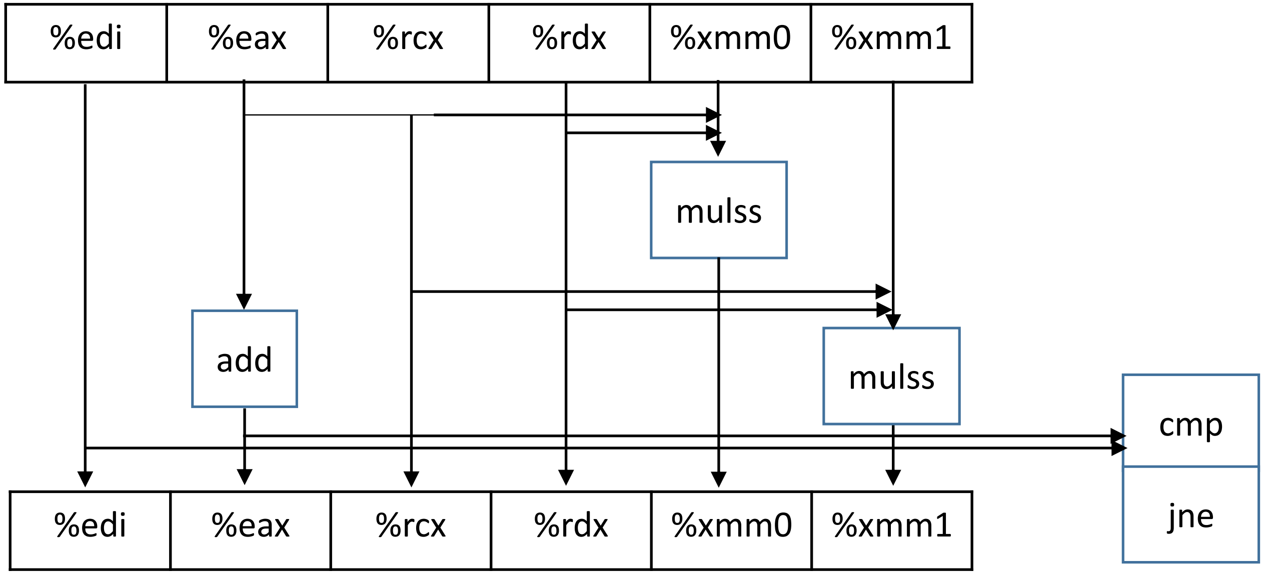
Combine6

```
400b27 <+28>:   movslq %eax,%rdx
400b2a <+31>:   mulss  (%rcx,%rdx,4),%xmm1
400b2f <+36>:   mulss  0x4(%rcx,%rdx,4),%xmm0
400b35 <+42>:   add    $0x2,%eax
400b38 <+45>:   cmp    %r8d,%eax
400b3b <+48>:   jl     0x400b27 <+28>
```

Flow diagram

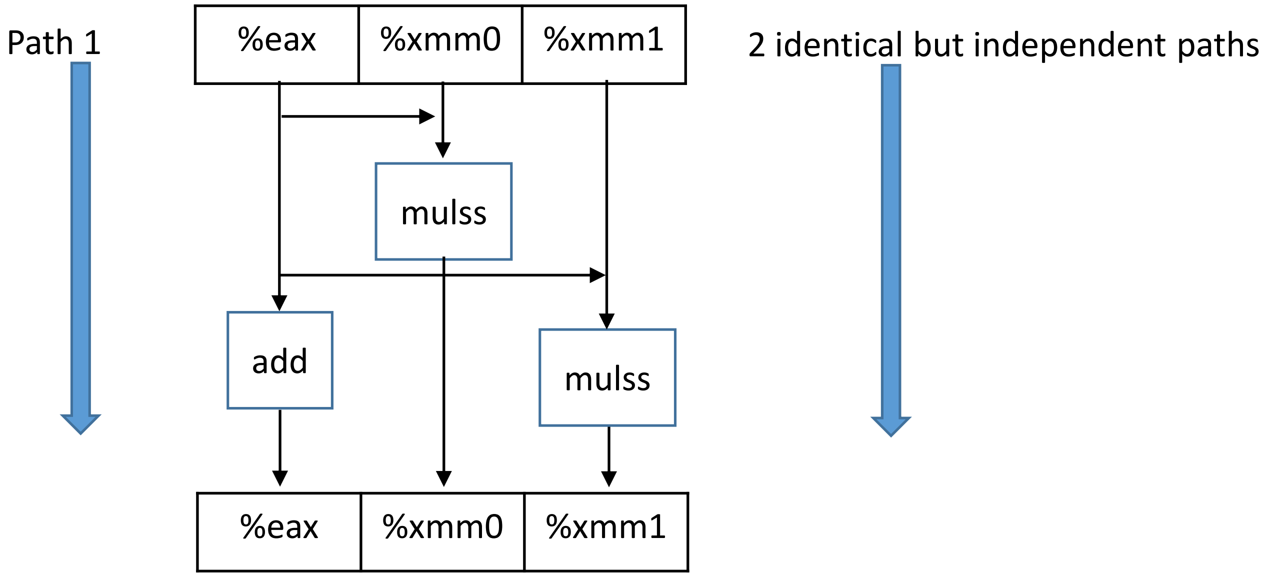
combine6 with OP = *, IDENT = 1

```
movslq %eax,%rdx
mulss  (%rcx,%rdx,4),%xmm1
mulss  0x4(%rcx,%rdx,4),%xmm0
add    $0x2,%eax
cmp    %r8d,%eax
jl     0x400b27 <+28>
```



Which operations depend on the outcome of the previous iteration of the loop?

Eliminate read only and local registers



Right two paths are critical with latency of 4: done half of the time

Reassociation

data_t is a data type: int or float or double.
IDENT is 1
OP is *

```
void combine7(vec_ptr v, data_t *dest)
  {
  int I, l = vec_length(v);
  data_t *data = get_vec_start(v) ;
  data_t  acc = IDENT ;

  for (i = 0; i < l     ; i+=2)
     acc = acc OP ( data[i] OP data[i+1] ) ;

  for(  ;   i<l ; i++ )
     acc = acc OP data[i] ;

  *dest = acc ;
  }
```

Timings for multiplication (OP = *, IDENT = 1, data_t = float)

Combine5 (unroll x 2)	Combine7 (unroll x 2 with reassociation)
Time= 0.1515 n= 100000	Time= 0.0771 n= 100000
Time= 0.3016 n= 200000	Time= 0.1511 n= 200000
Time= 0.4523 n= 300000	Time= 0.2266 n= 300000
Time= 0.6029 n= 400000	Time= 0.3019 n= 400000
Time= 0.7536 n= 500000	Time= 0.3777 n= 500000

Twice the speed. Similar to Combine6. Why?

Compiled code

Combine5

```
400b27 <+28>:   movslq %eax,%rcx
400b2a <+31>:   mulss  (%rdx,%rcx,4),%xmm0
400b2f <+36>:   mulss  0x4(%rdx,%rcx,4),%xmm0
400b35 <+42>:   add    $0x2,%eax
400b38 <+45>:   cmp    %edi,%eax
400b3a <+47>:   jl     0x400b27 <+28>
```

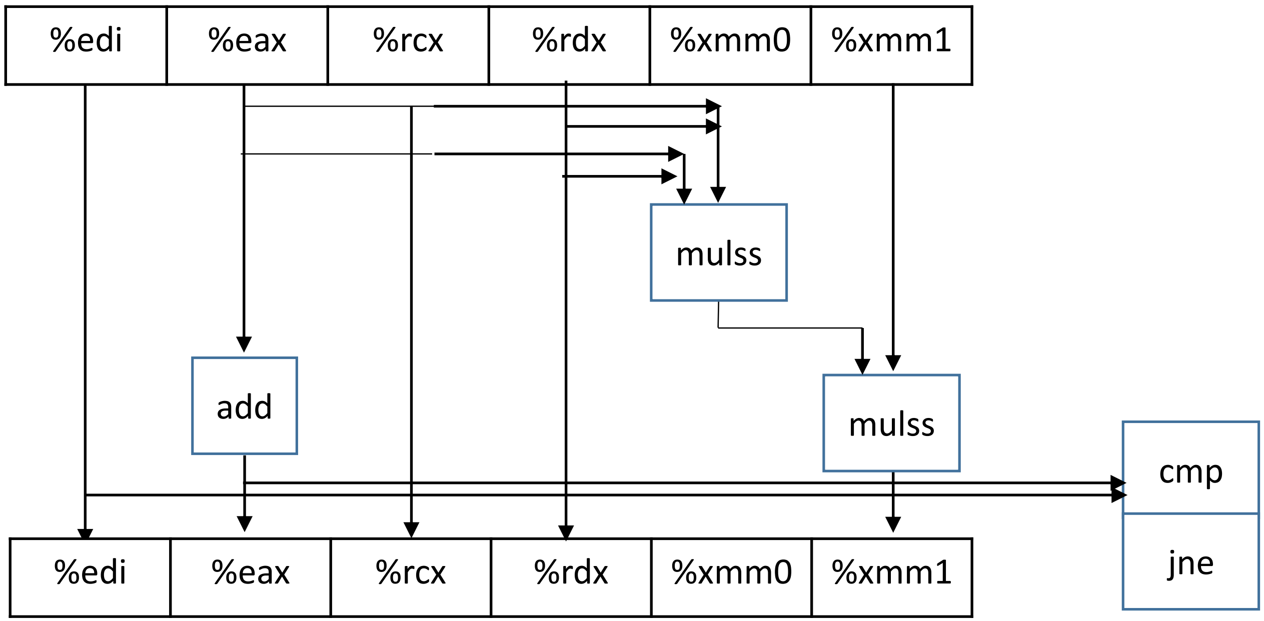
Combine7

```
400b27 <+28>:   movslq %eax,%rcx
400b2a <+31>:   movss  (%rdx,%rcx,4),%xmm1
400b2f <+36>:   mulss  0x4(%rdx,%rcx,4),%xmm1
400b35 <+42>:   mulss  %xmm1,%xmm0
400b39 <+46>:   add    $0x2,%eax
400b3c <+49>:   cmp    %edi,%eax
400b3e <+51>:   jl     0x400b27 <+28>
```

Flow diagram

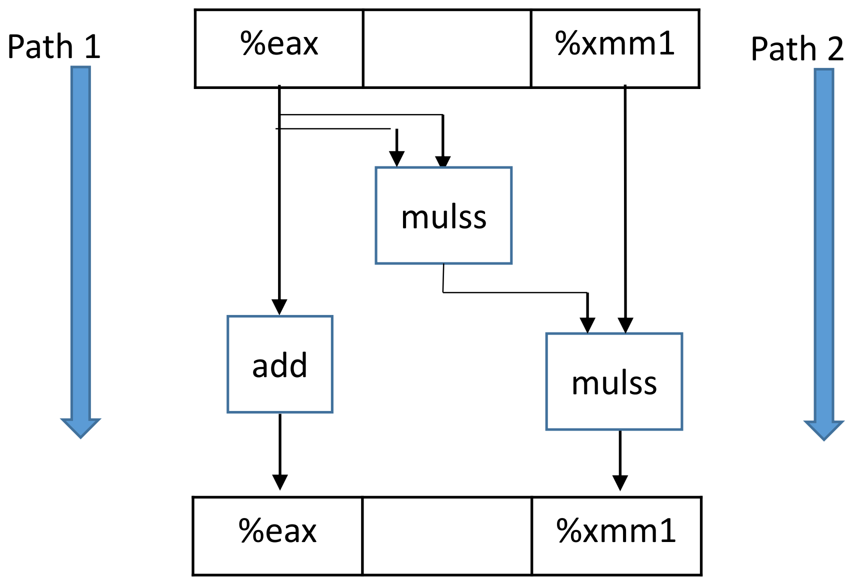
Combine7 with OP = *, IDENT = 1

```
movslq %eax,%rcx
movss  (%rdx,%rcx,4),%xmm1
mulss  0x4(%rdx,%rcx,4),%xmm1
mulss  %xmm1,%xmm0
add    $0x2,%eax
cmp    %edi,%eax
jl     0x400b27 <+28>
```



Which operations depend on the outcome of the previous iteration of the loop?

Eliminate read only and local registers



Path 2 is the critical path with latency of 4: done half of the time. The mulss in the middle can be done in parallel for the next iteration while the one on the right finishes for the previous.

Additional selected topics

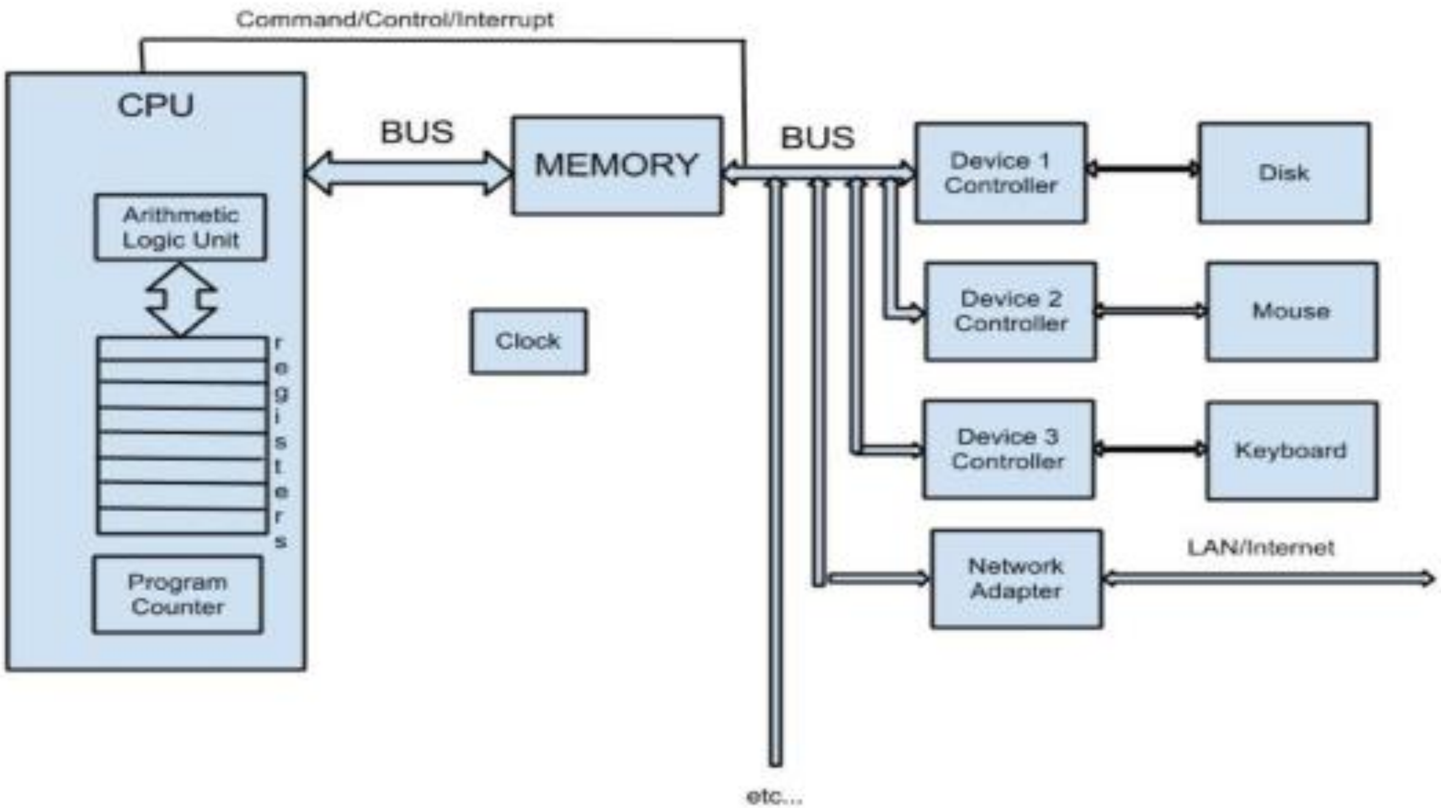
- Multiple accumulators limited by the number of registers
- Branch prediction errors can be minimized by using conditional moves
- Branch prediction errors can be affected by the data
- Reading from memory is usually slower than writing to memory (cache)
- Locate the fat by using a profiler

Technologies

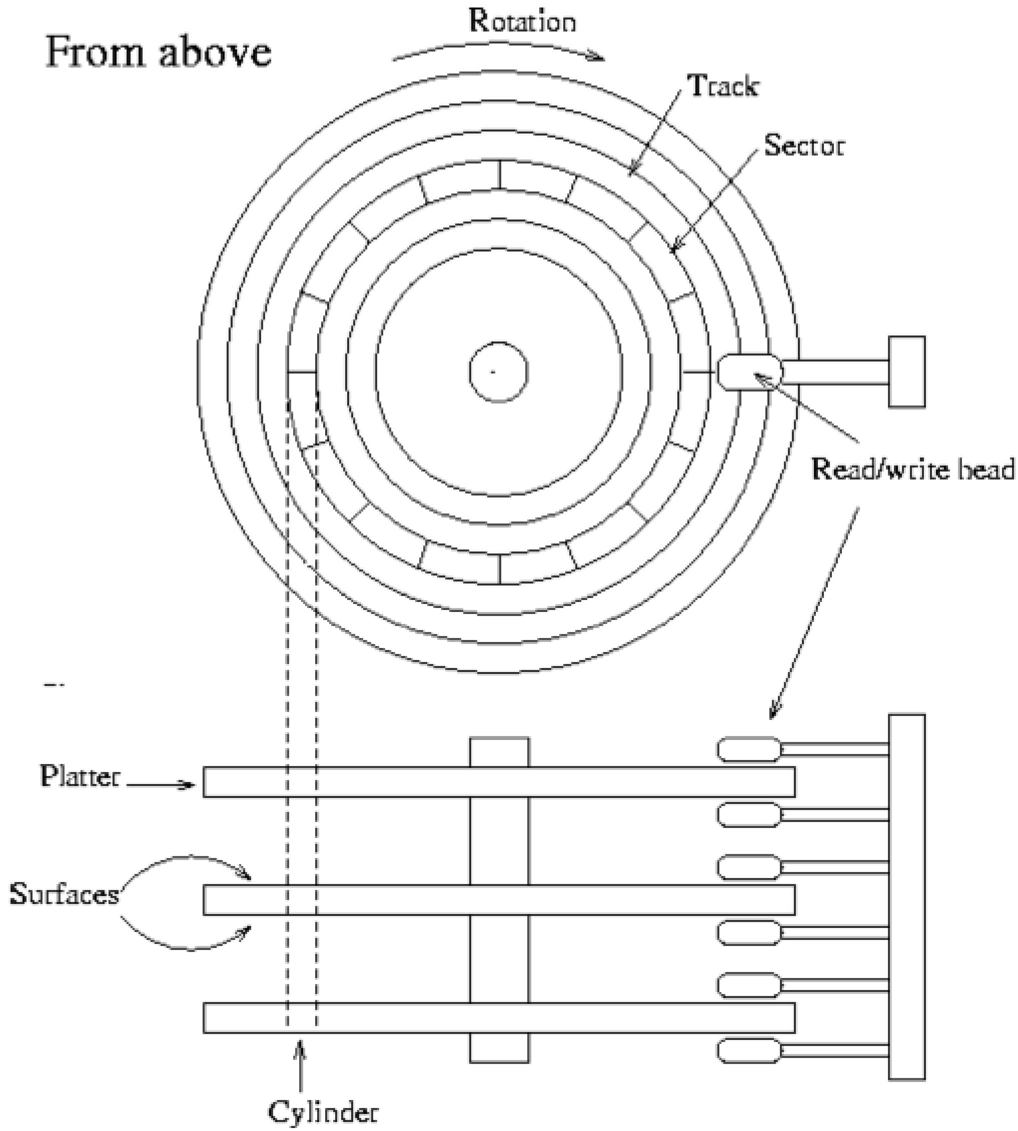
- Random Access Memories
- Disk Memories
- Solid State Disks

- Random Access Memories
    - Read only
        - Programmable
        - Erasable
        - Flash
    - SRAM Static -- fastest, more expensive, stable: caches
    - DRAM Dynamic -- slower, less expensive: main memory speed up by storing parts of words in parallel.
        - Lots of enhanced types

Overview of a Digital Computer



Disk Drive



Address on drive: cylinder (0-X), track (0-5), sector (0-Y)

Disk Drive Terminology

Recording density bytes/inch
Track density tracks/diameter
Areal density = (Track * Recording)

Capacity:

Bytes/sector
Sectors/track
Tracks/surface
Surfaces/platter
Platters/disk

Disk capacity is the product of these.

Speed:

Seek time
Rotational latency
Transfer time

Time to read is the sum of these.

Memory locality

Spatial
Temporal

```
int sumvec(int v[N])
  {
  int i, sum = 0;

  for (i = 0; i < N; i++)
    sum += v[i];
  return sum;
  }
```

Address	0	4	8	12	16	20	24	28
Contents	v0	v1	v2	v3	v4	v5	v6	v7
Access order	1	2	3	4	5	6	7	8

```
int sumarrayrows(int a[M][N])
  {
  int i, j, sum = 0;

  for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
      sum += a[i][j];
  return sum;
  }
```

Address	0	4	8	12	16	20	24	28
Contents	a00	a01	a02	a10	a11	a12	a20	a21
Access order	1	2	3	4	5	6	7	8

For N = 3

Memory locality

```
int sumarraycols(int a[M][N])
  {
  int i, j, sum = 0;

  for (j = 0; j < N; j++)
    for (i = 0; i < M; i++)
      sum += a[i][j];
  return sum;
  }
```

Accesses every Nth location in memory, M times. Bad spatial locality.

Program instruction display good spatial and temporal locality. Large loops interrupt the spatial and temporal locality.

Ways to improve locality

Consider src[n][n] and dst[n][n]. Both functions “rotate” the array by 90 degrees.

```
void rotate1( int *src, int *dst, int n )
 {
 int i, j;

 for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
     dst[(n-1-j)*n+i] = src[(i*n+j)] ;
 }
```

```
void rotate2(int *src, int *dst, int n) {
   int i, j;
   int ii, jj;

   for(ii=0; ii < n; ii+=8)
     for(jj=0; jj < n; jj+=8)
       for(i=ii; i < ii+8; i++)
         for(j=jj; j < jj+8; j++)
           dst[(n-1-j)*n+i] = src[(i)*n+j];
    }
```

This is called “blocking” to improve spatial locality.

Memory Hierarchy

Timings

rotate1	rotate2
Time= 0.0004 n= 16	Time= 0.0009 n= 16
Time= 0.0010 n= 32	Time= 0.0032 n= 32
Time= 0.0041 n= 64	Time= 0.0099 n= 64
Time= 0.0448 n= 128	Time= 0.0338 n= 128
Time= 0.2497 n= 256	Time= 0.0876 n= 256
Time= 1.3555 n= 512	Time= 0.3815 n= 512
Time= 5.5756 n= 1024	Time= 2.9030 n= 1024
Time= 76.0182 n= 2048	Time= 26.7598 n= 2048

Memory

Haber's locality figure of merit
min = moving min of 8 items, same for max
min-max = absolute value min-max

Partial list for n = 16

Rotate1

i	j	(n-1-j)*n+i	(i*n+j)	min	max	min-max
0	0	240	0			0
0	1	224	1			0
0	2	208	2			0
0	3	192	3			0
0	4	176	4			0
0	5	160	5			0
0	6	144	6			0
0	7	128	7	128	240	112
0	8	112	8	112	224	112
0	9	96	9	96	208	112
0	10	80	10	80	192	112
0	11	64	11	64	176	112
0	12	48	12	48	160	112
0	13	32	13	32	144	112
0	14	16	14	16	128	112
0	15	0	15	0	112	112
1	0	241	16	0	241	241
1	1	225	17	0	241	241
1	2	209	18	0	241	241
1	3	193	19	0	241	241
1	4	177	20	0	241	241
1	5	161	21	0	241	241
1	6	145	22	0	241	241
1	7	129	23	129	241	112
1	8	113	24	113	225	112
1	9	97	25	97	209	112
1	10	81	26	81	193	112
1	11	65	27	65	177	112
1	12	49	28	49	161	112
1	13	33	29	33	145	112

Rotate2

i	j	(n-1-j)*n+i	(i*n+j)	min	max	min-max
0	0	240	0			
0	1	224	1			
0	2	208	2			
0	3	192	3			
0	4	176	4			
0	5	160	5			
0	6	144	6			
0	7	128	7	128	240	112
1	0	241	16	128	241	113
1	1	225	17	128	241	113
1	2	209	18	128	241	113
1	3	193	19	128	241	113
1	4	177	20	128	241	113
1	5	161	21	128	241	113
1	6	145	22	128	241	113
1	7	129	23	129	241	112
2	0	242	32	129	242	113
2	1	226	33	129	242	113
2	2	210	34	129	242	113
2	3	194	35	129	242	113
2	4	178	36	129	242	113
2	5	162	37	129	242	113
2	6	146	38	129	242	113
2	7	130	39	130	242	112
3	0	243	48	130	243	113
3	1	227	49	130	243	113
3	2	211	50	130	243	113
3	3	195	51	130	243	113
3	4	179	52	130	243	113
3	5	163	53	130	243	113

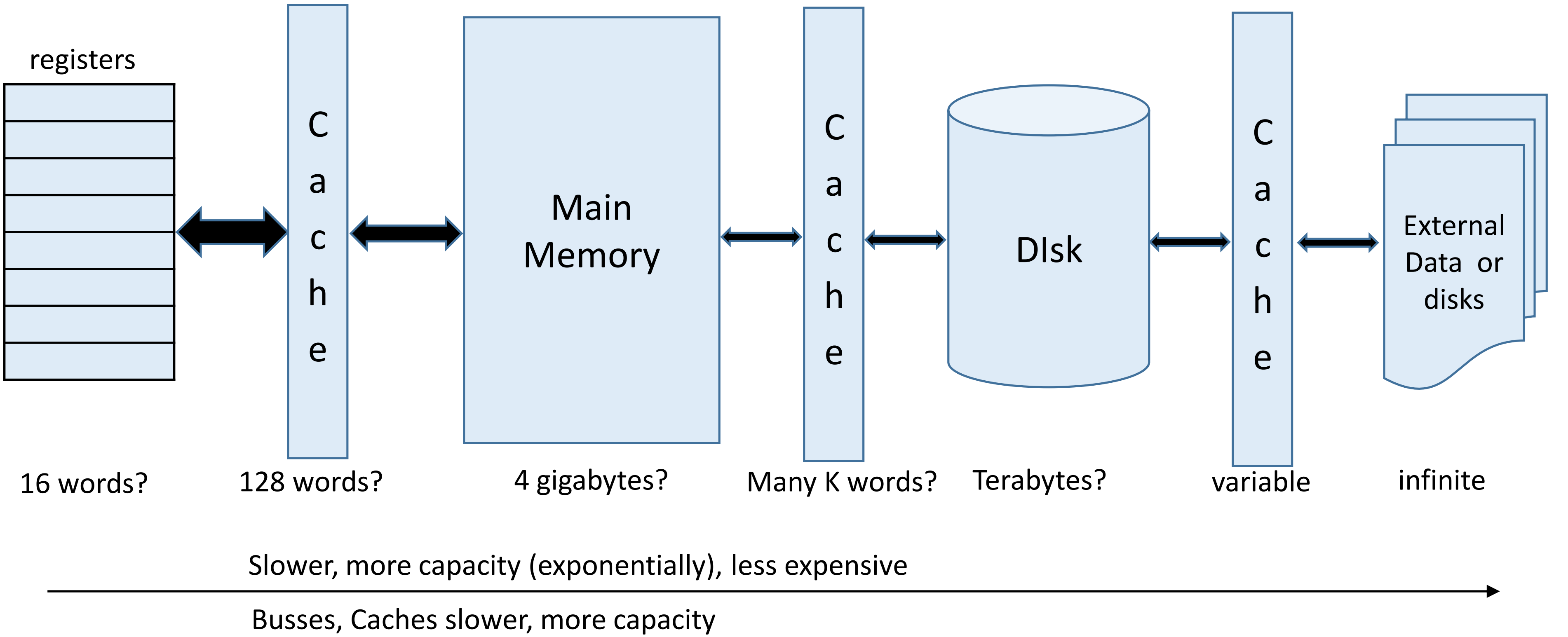
Haber's locality figure of merit
Average of min – max

rotate1	
n= 16	166.40
n= 32	388.08
n= 64	834.64
n= 128	1729.83
n= 256	3521.40
n= 512	7105.20
n= 1024	14272.93
n= 2048	30051.08

rotate2	
n= 16	116.81
n= 32	241.23
n= 64	492.40
n= 128	995.97
n= 256	2003.17
n= 512	4018.24
n= 1024	8050.08
n= 2048	17160.27

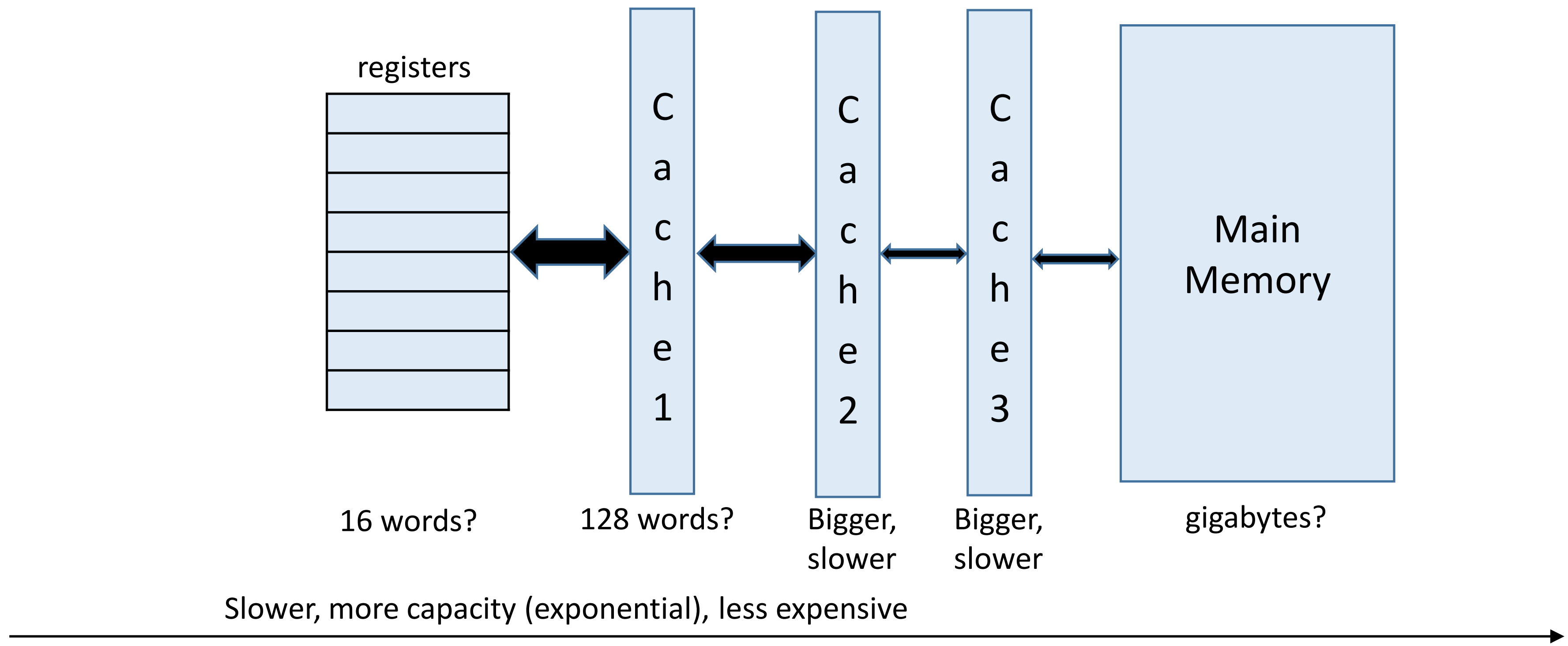
Memory Hierarchy

Faster to slower, more expensive to cheaper, less capacity to greater



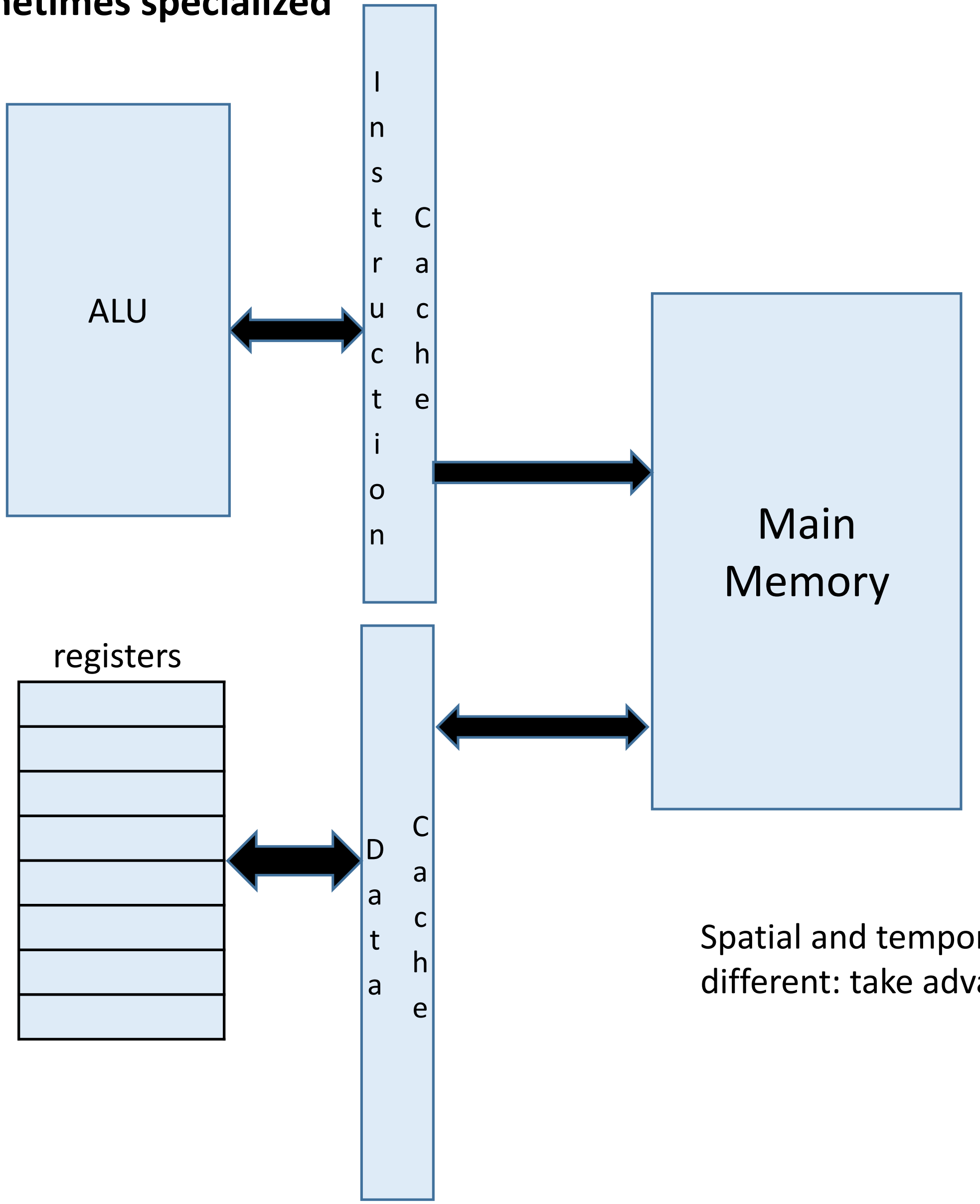
Memory Hierarchy

Caches are sometime cascaded



Memory Hierarchy

Caches are sometimes specialized



Spatial and temporal locality are grossly different: take advantage

The cache principle

- An intermediary between a slower and faster memory
- A device with memory and an autonomous program
- Speed up the memory access

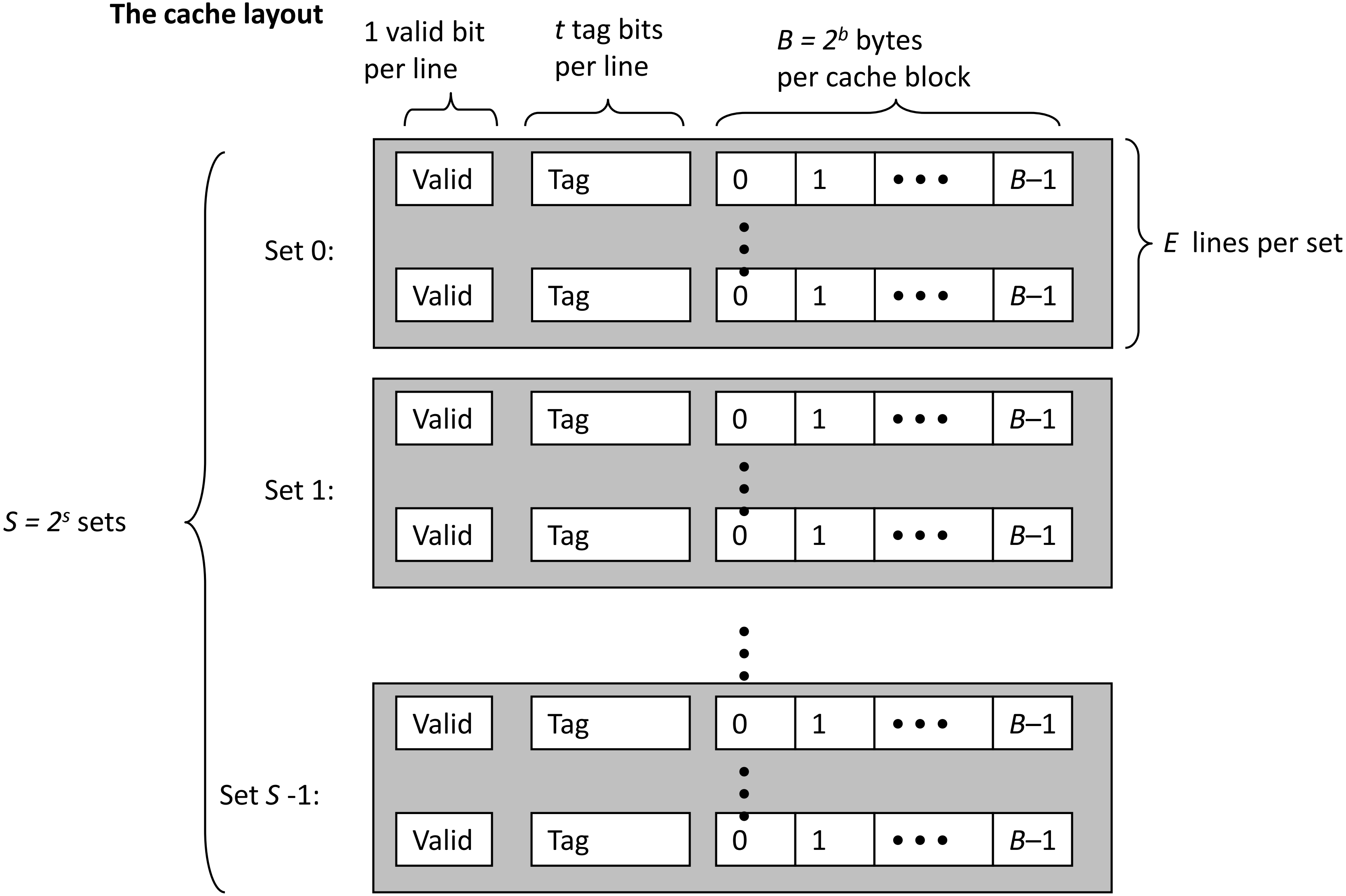
M = 2^m bytes is the main memory size (m bit unsigned integer address)
Divide slower memory into fixed size blocks (usually small) B = 2^b
Build cache with with a certain number of sets S = 2^s each containing a B sized block
Some caches have a certain number of lines per set: E. A special case is where E = 1, call direct mapped cache.

The cache size is B x E x S = C bytes. The cache “covers” C/M of the memory

Each set has a tag which is t bits.

Cache Memory

The cache layout



Cache size: $C = B \times E \times S$ data bytes

Summary of cache layout

M = 2^m bytes is the main memory size (m bit unsigned integer address)

The cache has

S = 2^s sets
E lines per set
B = 2^b block size in bytes

Each set has a tag which is m bits long

The tag is divided into 3 parts: the block offset, the set index and the tag.
Since the tag is a total of m bits then:

$$t = m - s - b$$

A m bit memory address is divided into 3 parts the same way:

t bits	s bits	b bits
tag	set index	block offset

This means that each 2^t size memory block is mapped into the same cache area.

A 32 bit memory address example

t	t	t	t	t	t	t	t	s	s	s	s	s	s	s	s	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b	b
31								23								15															0

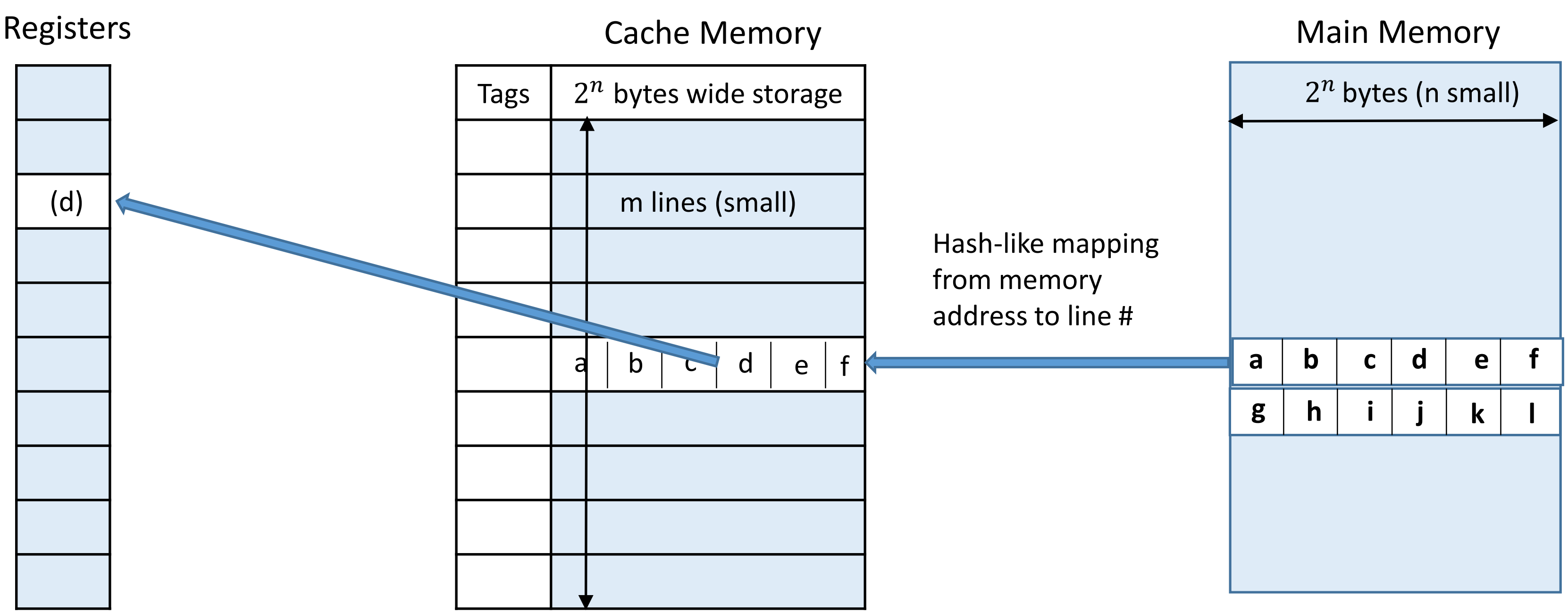
and in this case, we have t = 8, s = 8 and b = 16. This is a huge cache!

B = 65,536
S = 256

The cache principle

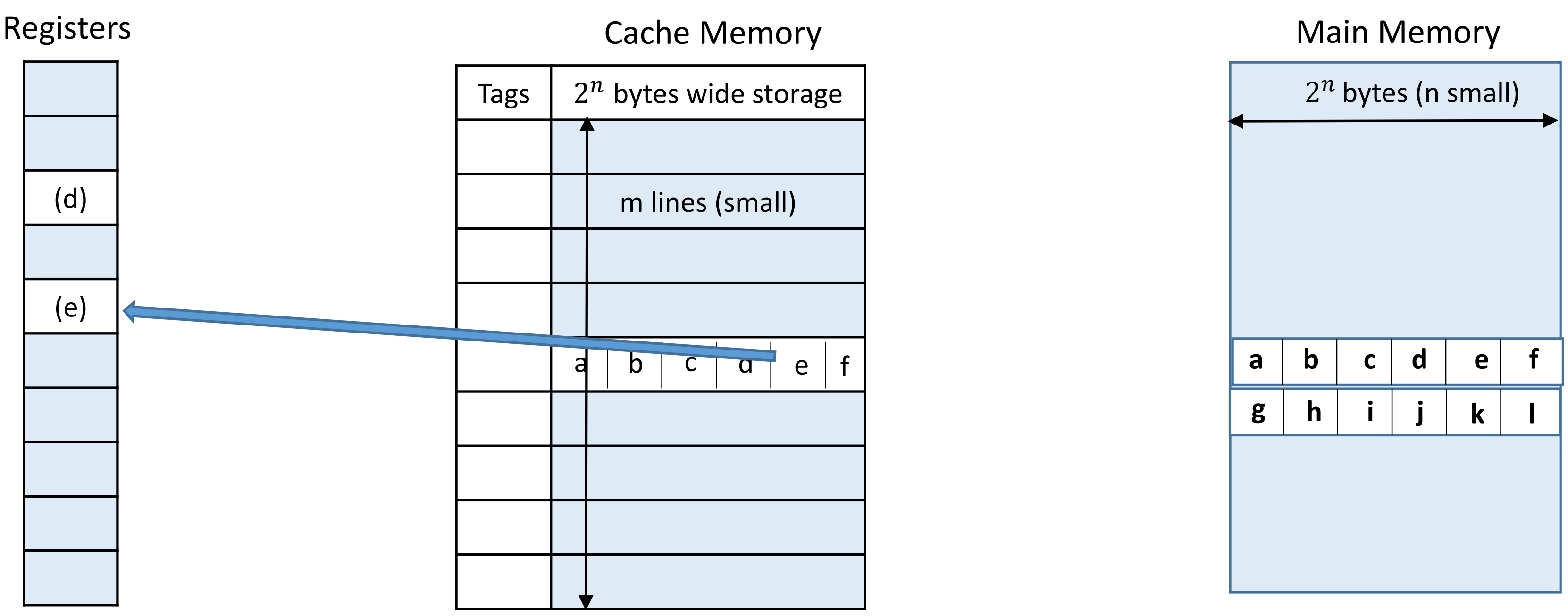
- An intermediary between a slower and faster memory
- A device with memory and an autonomous program
- Speed up the memory access

Clean start : retrieve(address (d), bytes (1, 2, 4 or 8))



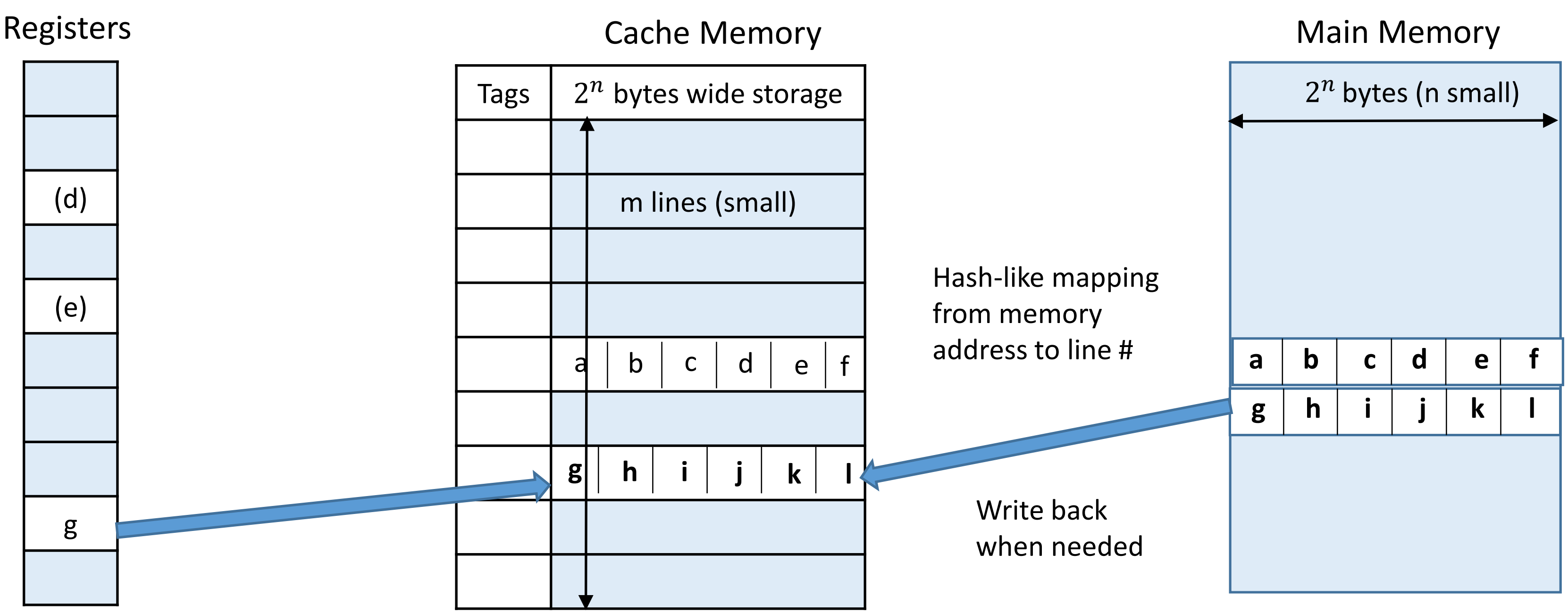
The cache principle

Next Instruction: retrieve(address (e), bytes (1, 2, 4 or 8))



The cache principle

Next Instruction: write(address (g), bytes (1, 2, 4 or 8))



Summary of Cache Operation

Read from Memory			
	Item in Cache ?		
		Read Hit :	Return item
		Read Miss :	Select cache line Read item and surroundings into cache Return item
Write to memory			
	Item in Cache ?		
		Write Hit:	Write item into cache
		Write Miss :	Select cache line Read item and surroundings into cache Write item into cache

Select Cache Line: Map memory address to cache line number using hash-like formula

	Cache line in use?	No: Finished
	Yes:	Correct memory segment? Yes: finished No: Dirty bit set? : Write line from cache to memory

Cache sample: direct mapped -> E = 1

lets try a cache like this:

```
#define S      4   // (4 cache sets)
#define E      1  // (direct mapped cache)
#define B      16  // (16 elements in each block)
#define T      2   // (2 tag bits)
#define M      256 // (256 byte memory)

struct cache_t
  {
  char valid ;    // simulate a bit with this
  char dirty ;    // an element was stored but not written
  int tag   ;
  char *block ;
  } cache[S] ;

 char memory[M] ;
```

We would initialize the cache by:

```
  for( i=0; i<S; i++ )
    {
    cache[i].block = (int *) malloc(B) ;
    cache[i].valid = 0 ;
    }
```

and

```
  int s = log2(S) ;
  int b = log2(B) ;
  int m = log2(M) ;
```

Memory retrieval

Retrieve something from memory address: a.

int *si* = (a >> m-T-s) % S ;	the set index (m-T-s = b)
int *ta* = a >> m-T ;	the tag
int *bo* = a % B ;	the block offset
char hit ;	

Example: retrieve from address 152

ta		si		bo			
1	0	0	1	1	0	0	0

ta = 2 ;

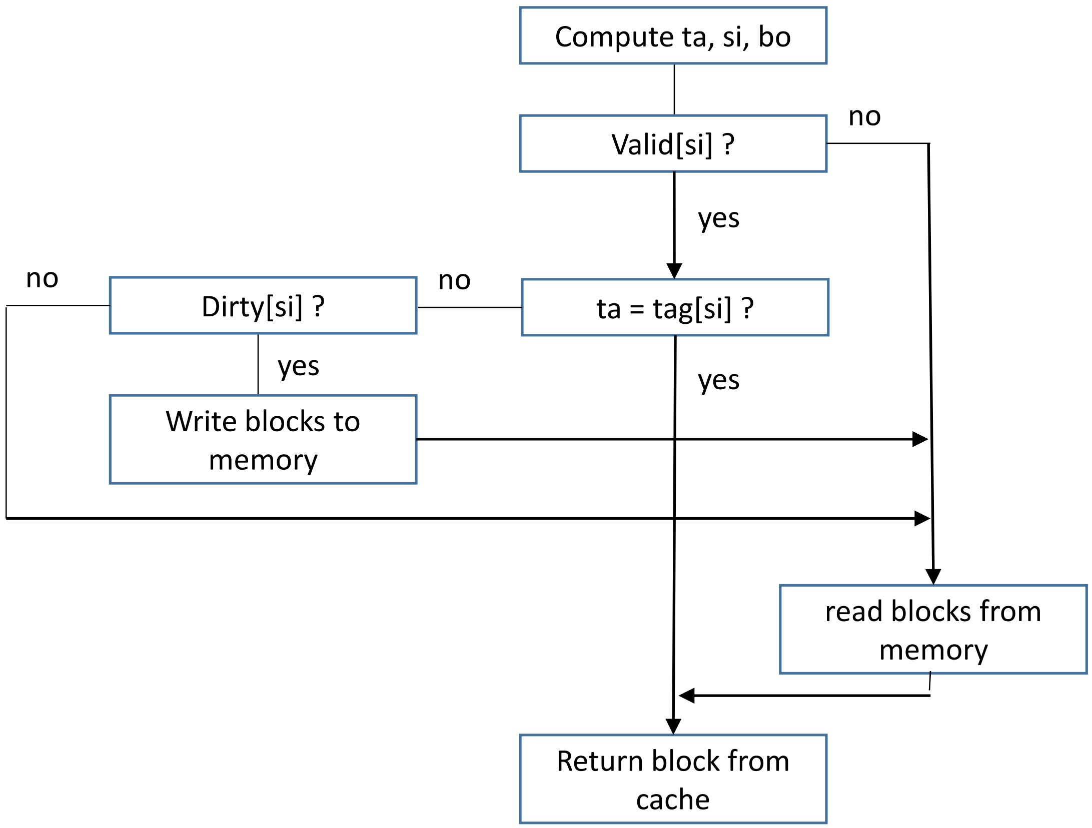
si = 1 ;

bo = 8 ;

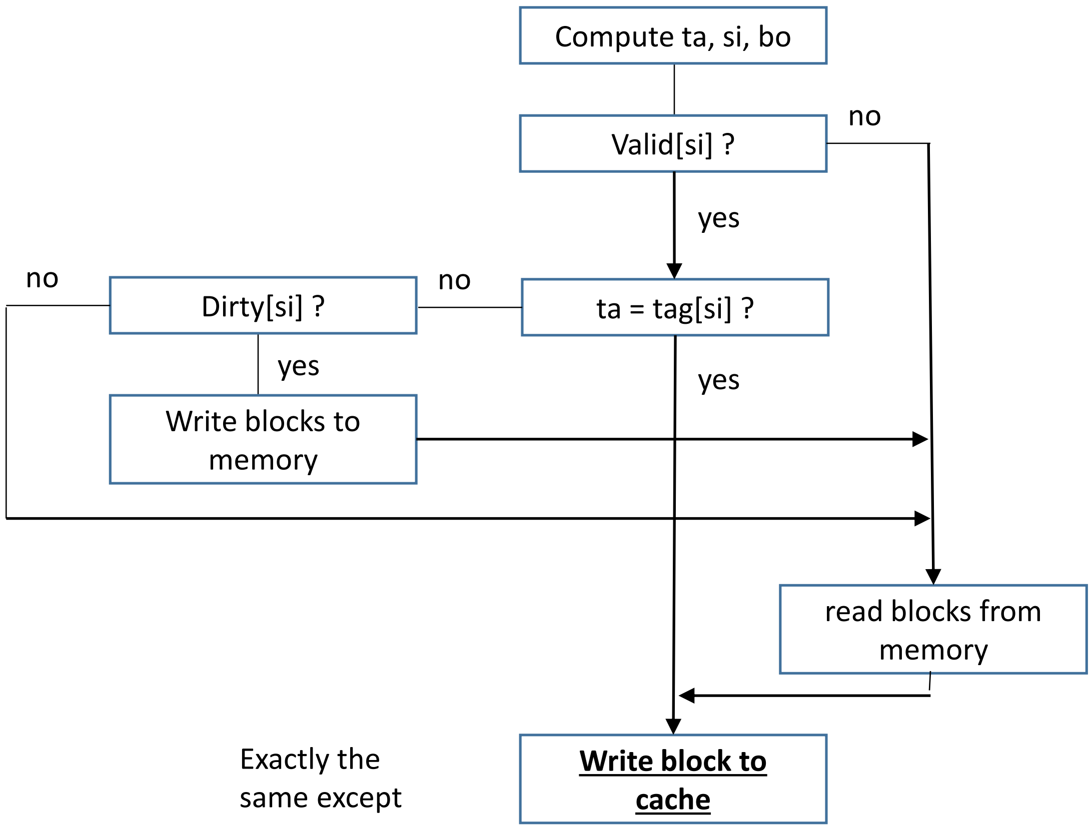
Memory retrieval

```
If ( cache.valid[si] )
  if ( ta == cache.tag[si]  >> m-T )         // the tag
    hit = 1 ;
  else
     {
     if ( cache.dirty[si] )
       for( i = (cache.tag[si] >> b) << b, j = 0; j < B; i++,  j++ )  // writes the cache to memory
         memory[i] = cache.block[si][j] ;
      hit = 0 ;
else
  hit = 0 ;

if ( ~hit )
  {
  for( i = ( a >> b ) <<b, j = 0; j < B; i++, j++ )
    cache.block[si][j] = memory[i] ;
  cache.valid[si] = 1 ;
  cache.dirty[si] = 0 ;
  cache.tag[si] = a ;
  }

return cache.block[si][bo] ;
```

Memory retrieval flowchart



Write to memory flowchart

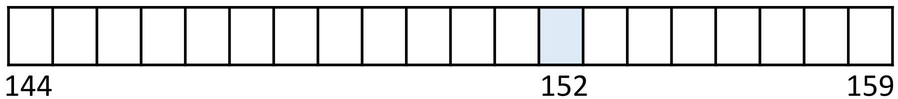


Exactly the same except

Memory retrieval

If hit = 0, we have to go to memory

get from a - a % B   to   a - a % B + B




```
If ( ~hit )
  {
  for( i = a - a % B, j=0 ; i < a - a % B + B ; i++,j++ )
    cache.block[si][j] = memory[i] ;

   cache.valid[si] = 1 ;
   cache.ta[si] = a ;
   }
```

We retrieve block bo from set si

*a = cache.block[si][bo] ;

cache.c

103

ta		si		bo			
0	1	1	0	0	1	1	1
1		2		7			

Miss! Set 2 now contains 96 – 111   bo of 103 = 7

198

ta		si		bo			
1	1	0	0	0	1	1	0
3		0		6			

Miss! Set 0 now contains 192 – 207 bo of 196 = 6

105

ta		si		bo			
0	1	1	0	1	0	0	1
1		2		9			

Hit! Set 2 now contains 96 – 111 bo = 9

236

ta		si		bo			
1	1	1	0	1	1	0	0
3		2		12			

Line replacement miss!
Set 2 contains 96 – 111, throw it out, replace with 224 - 239

running cache.c with stride = 1

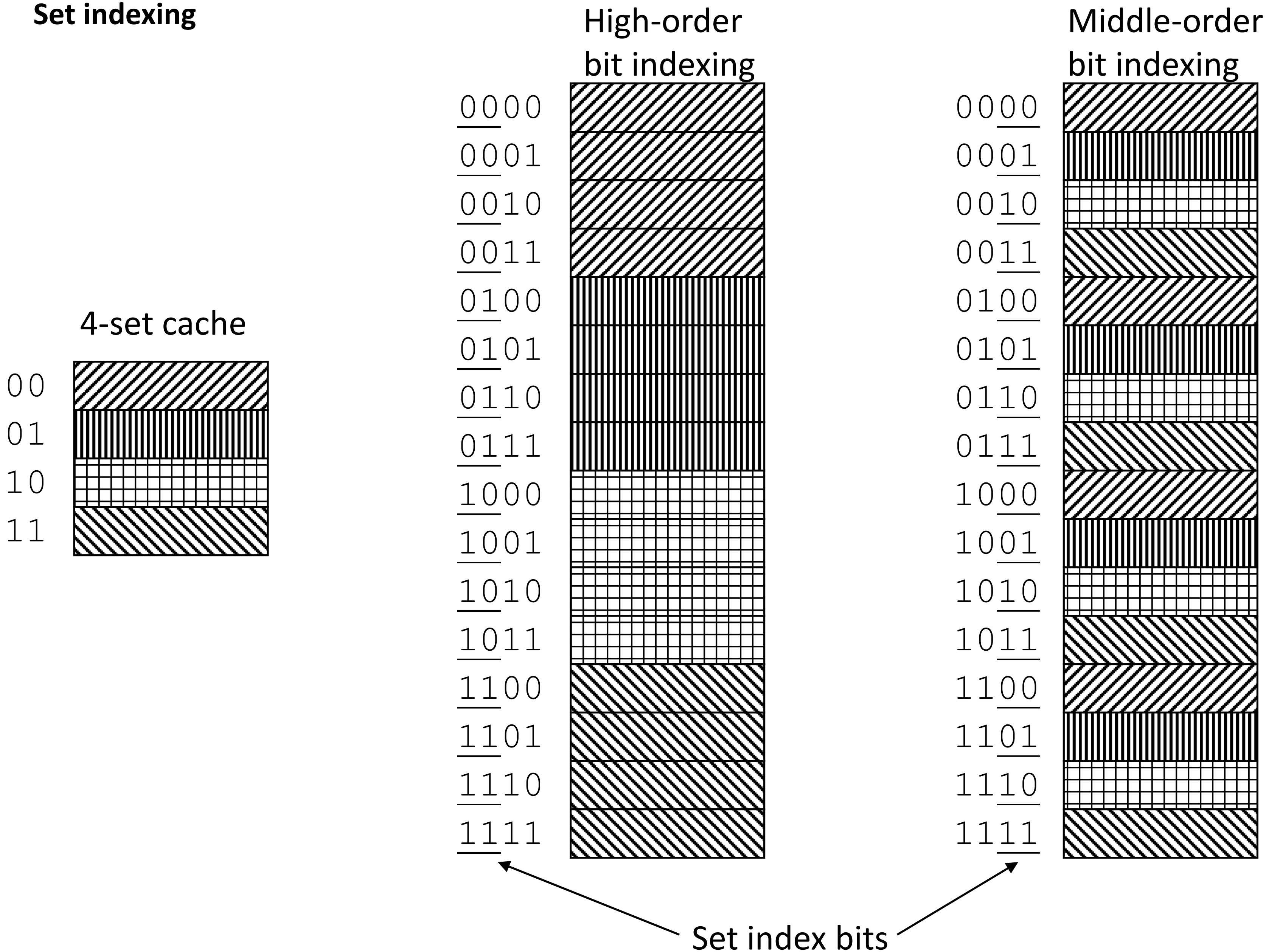
```
for( i=0; i<256; i++ )
  read from memory
```

We get (ta,si,bo) = $\overbrace{\underbrace{(0,0,0)\ \ (0,0,1)\ \ldots\ (0,0,15)}_{\text{one set}}\ (0,1,0)\ (0,1,1)\ \ldots\ (0,3,15)}^{\text{one quadrant}}\ (1,0,0)\ \ldots\ (1,3,15)\ (2,0,0)\ \ldots\ (3,3,15)$

Looking at ta and si, it has mapped all of the possible M memory addresses onto the S sets and each set is identified by the tag. t concatenated with s is all of the addresses in memory divided by B.

In our case, the tag denotes which “quadrant” of memory, 0-63, 64-127, 128-191, 192-255. The set index, divides each quadrant into quadrants: 0-15, 16-31, 32-47, 48-63.

Set indexing



Powers of two problems

Arrays separated by powers of 2 addresses.

In previous example, the memory is in quadrants.

char a[64], b[64] and address of a = 0 and address of b = 0

```
for( i=0; i<64; i++ )
   b[i] = a[i] ;
```

	i	addr	ta	si	Bo
access a	0	0	0	0	0
access b	0	64	1	0	0
access a	1	1	0	0	1
access b	1	65	1	0	1

Known as cache thrashing

Associative Caches, E > 1

```
#define S       4   // (4 cache sets)
#define E       2  // (direct mapped cache)
#define B       16  // (16 elements in each block)
#define T       2   // (2 tag bits)
#define M       256 // (256 byte memory)

struct cache_t
  {
  char valid ;    // simulate a bit with this
  char dirty ;    // an element was stored but not written
  int tag   ;
  char *block ;
  } cache[S][E] ;
```

Recall that C = E x B x S

Two types of associative caches:

1 < E < C/B	E way associative
E = C/B	fully associative

To convert from a direct cache E=1 to an associative cache of the same size with E > 1, divde the number of sets by 2^{E-1}

Associative Cache Example, E =3

aa

	valid	tag	block
set 1			

	valid	tag	block
set 2			

⋮

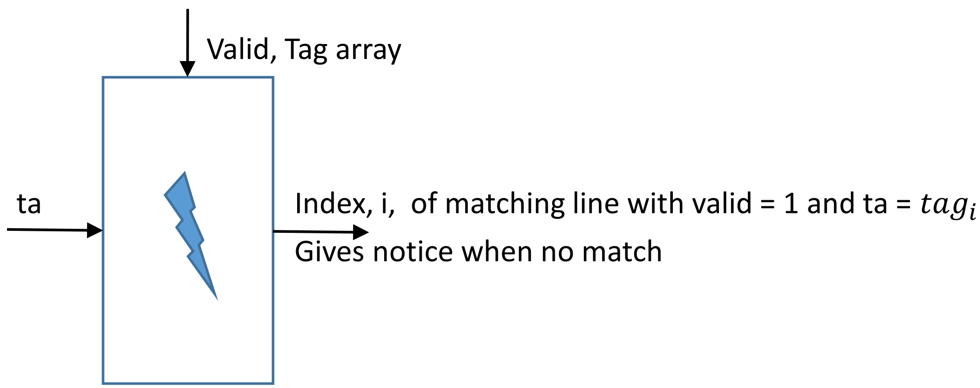
	valid	tag	block
set S-1			

Works the same, except

Can store blocks from different quadrant (same offset) in one set. Identified by Tag.

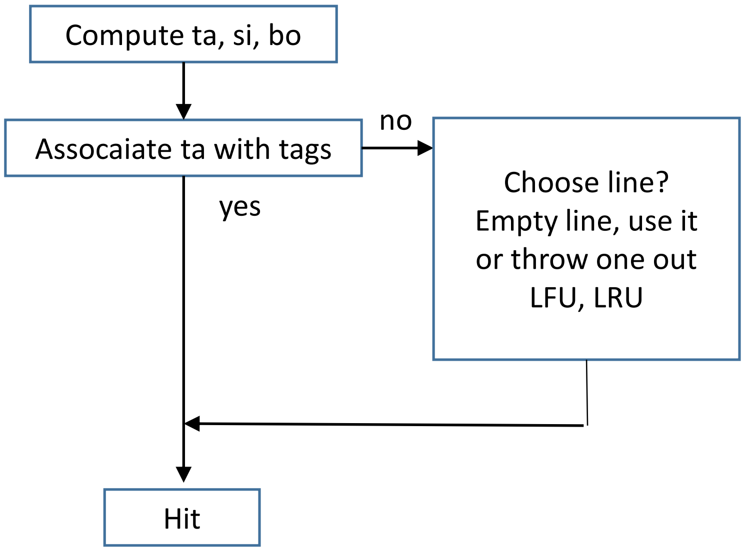
	valid	tag	block
	1	tag_1	block in quadrant tag_1
set x	0		
	1	tag_2	block in quadrant tag_2

To find which line: associative hardware:



Determine if hit?

Extra step in process when no hit



Fully Associative Caches, E = C/B

Only one set, C/B lines in set.

Recall:

t bits	s bits	b bits
tag	set index	block offset

But now s = 0

t bits	b bits
tag	block offset

Everything works the same

Write strategies

What to do with a write hit.

- write-through: write the set to memory immediately. High bus traffic can be done in background
- write-back: delay write to memory, requires a dirty bit, complicated replacement algorithm

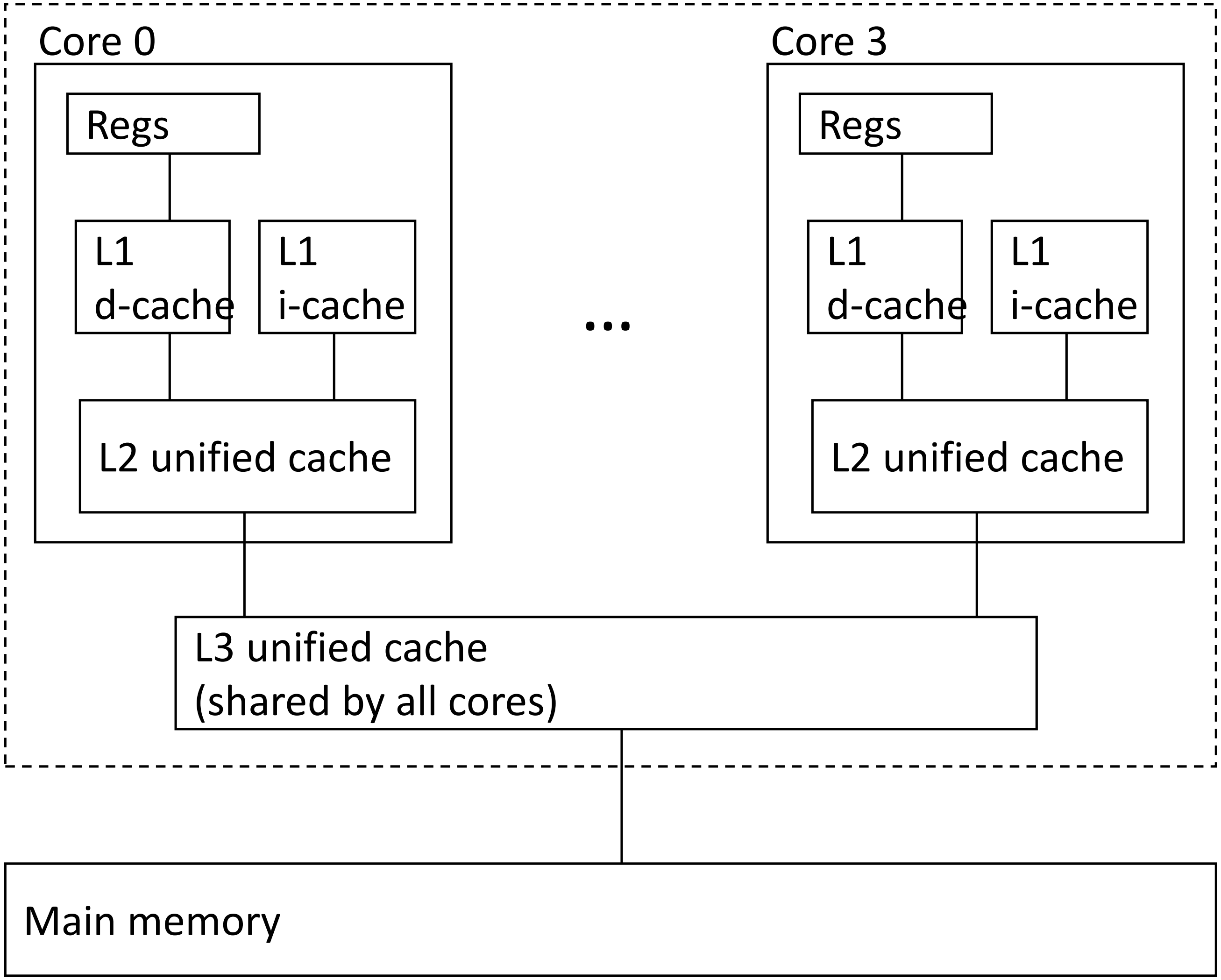
What to do with a write miss (block not in cache when storing)

- write-allocate: read block from lower level, then update
- write-no allocate

Most caches are write-back and write-allocate

Note that an instruction cash is a no write cache.

Intel Core I7 cache architecture



Intel Core I7 cache sizes and speeds

Cache type	L1 i-cache	L1 d-cache	L2 unified cache	L3 unified cache
Access time (cycles)	4	4	11	30-40
Cache size C	32KB	32KB	256KB	8MB
Assoc. (E)	8	8	8	16
Block size (B)	64 b	64 b	64 b	64 b
Sets (S)	64	64	512	8192

Cache Memory

Cache performance metrics:

- Miss rate. #misses/#references. Separate into read/write
- Hit rate. 1-miss rate.

- Hit time. The time to deliver a word in the cache to the CPU.
- Miss penalty Any additional time required because of a miss. (get from next level)

Cache performance factors:

- larger cache size: increased hit rate more memory but slower
- larger block size: lessens the impact of large working set. Requires more time to retrieve and store from lower level
- associativity (size of E): reduces thrashing probability but costly in $ due to complexity and increased record keeping in each line.
- write back strategy: write through is simple, reduces miss penalty

Cache friendly code

Good spatial locality

Common sense: look for the fat.

Example 1:

```
int sumvec(int v[N])
 {
 int i, sum = 0;

 for (i = 0; i < N; I++)
    sum += v[i];

return sum;
 }
```

With B = 4

v[i]	i = 0	1	2	3	4	5	6	7
Access order, [h]it or [m]iss	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]

Cache friendly code

Stride-k block size B

miss rate = min (1, (wordsize × k)/B)

Example wordsize = 1, k = 1, B = 30. Miss rate = 1/30, one miss every 30 references

Double the word size, double the miss rate. Each access "uses up" wordsize element in the block.

Another example

```
int sumarrayrows(int a[M][N])
{
int i, j, sum = 0;

for (i = 0; i < M; i++)
  for (j = 0; j < N; j++)
     sum += a[i][j];
return sum;
}
```

Still stride 1 because of ordering of arrays in memory

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	1 [m]	2 [h]	3 [h]	4 [h]	5 [m]	6 [h]	7 [h]	8 [h]
i = 1	9 [m]	10 [h]	11 [h]	12 [h]	13 [m]	14 [h]	15 [h]	16 [h]
i = 2	17 [m]	18 [h]	19 [h]	20 [h]	21 [m]	22 [h]	23 [h]	24 [h]
i = 3	25 [m]	26 [h]	27 [h]	28 [h]	29 [m]	30 [h]	31 [h]	32 [h]

Follows the miss rate formula.

Bad example

```
int sumarraycols(int a[M][N])
{
int i, j, sum = 0;

for (j = 0; j < N; j++)
   for (i = 0; i < M; I++)
       sum += a[i][j];
return sum;
}
```

Now stride 8 because of ordering of arrays in memory

a[i][j]	j = 0	1	2	3	4	5	6	7
i = 0	1 [m]	5 [m]	9 [m]	13 [m]	17 [m]	21 [m]	25 [m]	29 [m]
i = 1	2 [m]	6 [m]	10 [m]	14 [m]	18 [m]	22 [m]	26 [m]	30 [m]
i = 2	3 [m]	7 [m]	11 [m]	15 [m]	19 [m]	23 [m]	27 [m]	31 [m]
i = 3	4 [m]	8 [m]	12 [m]	16 [m]	20 [m]	24 [m]	28 [m]	32 [m]

Follows the miss rate formula: = 1

Figure 6.43

```
for (i = 0; i < elems; i += stride)
    {
    result += data[i];
    }
 sink = result; /* So compiler doesn't optimize away the loop */
```

Add up an array of size elems in stride “stride”. Caclulate the running time.

Figure 6.43

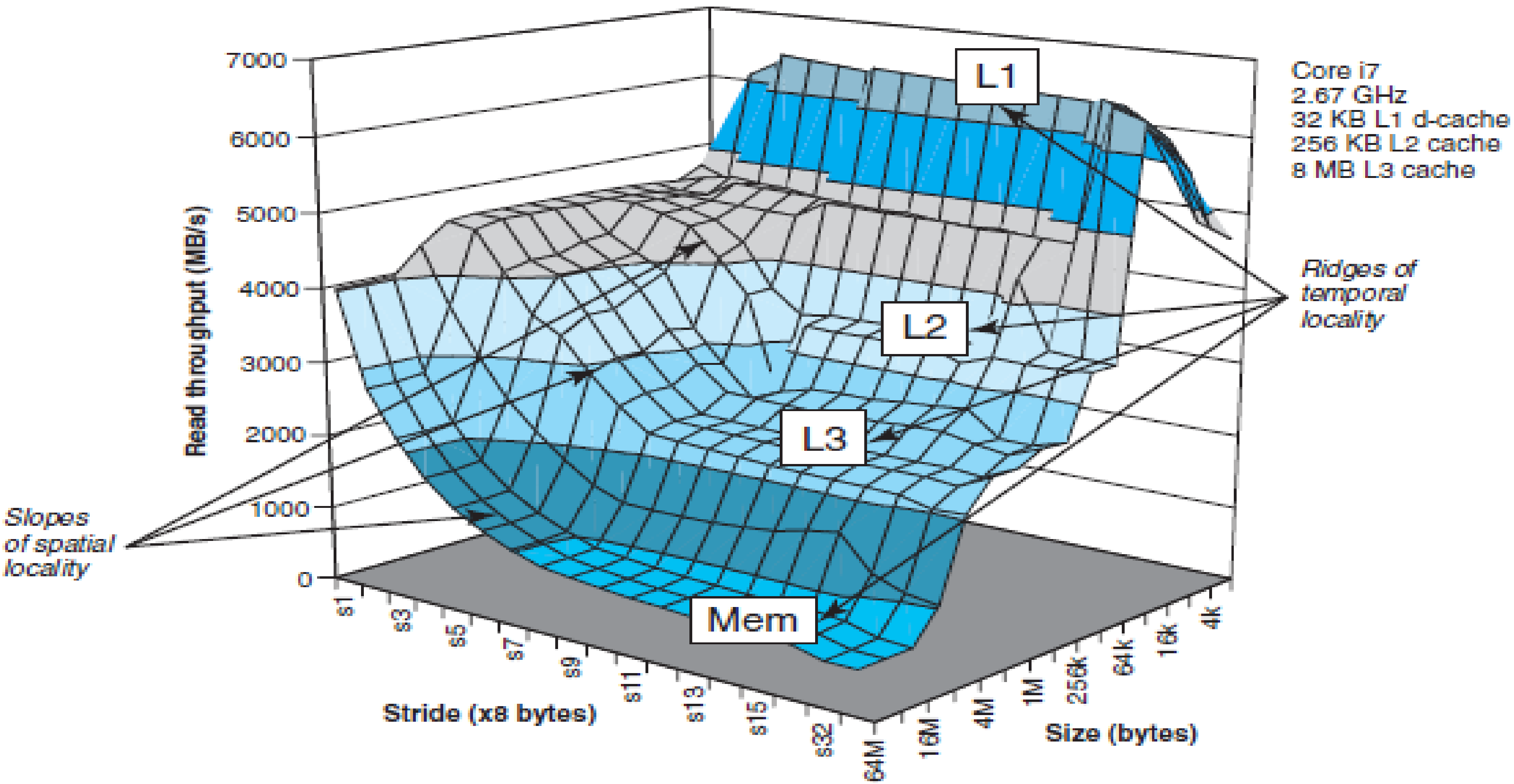
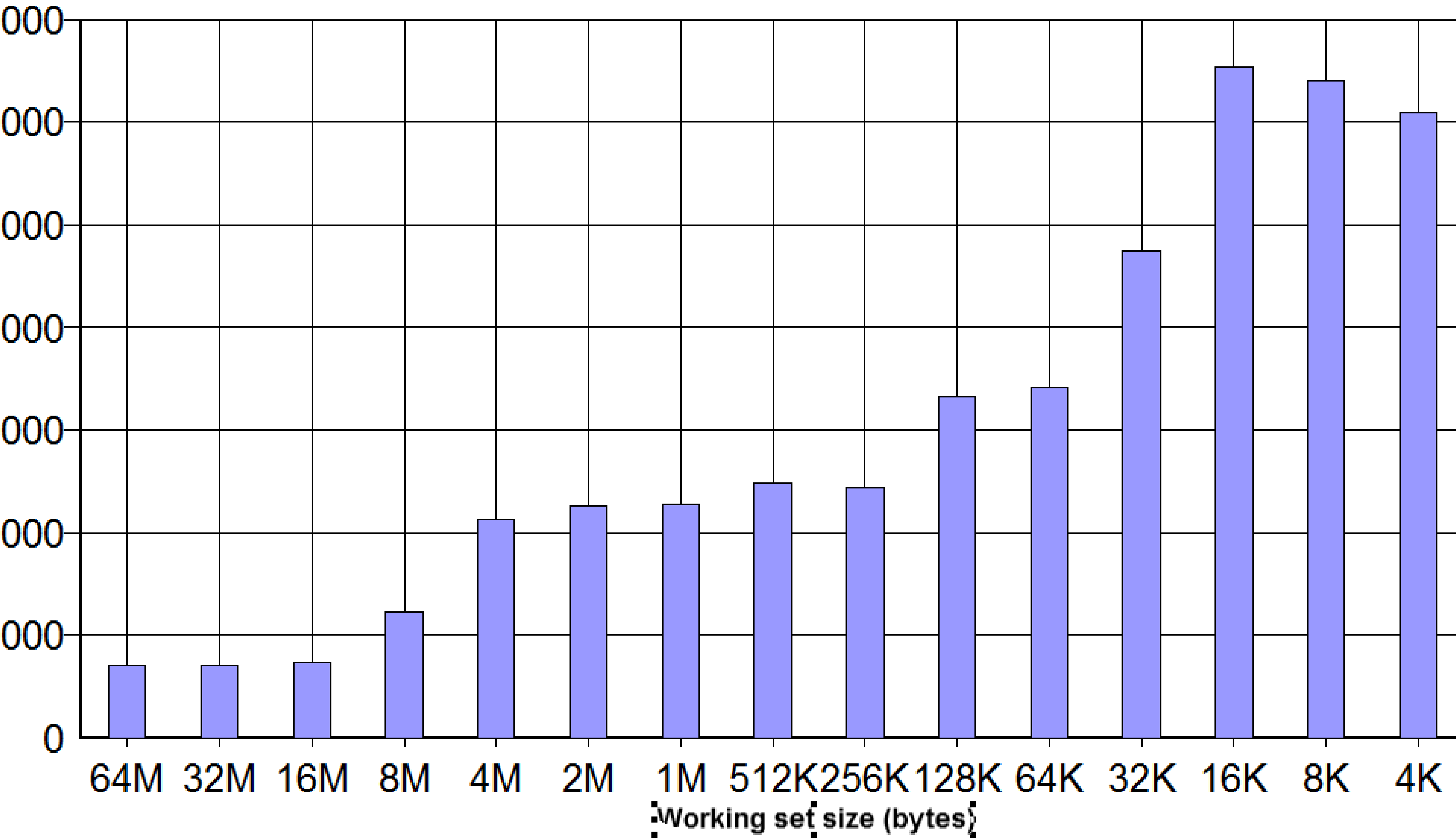


Figure 6.43 **The memory mountain.**

Figure 6.44


000
000
000
000
000
000
000
0
64M
32M
16M
8M
4M
2M
1M
512K
256K
128K
64K
32K
16K
8K
4K
Working set size (bytes)

Midterm Topics

November 17, 2014 Exam

Topics:

- Assembly Language
    - Arithmetic, logical and control instructions
    - Procedure calls, stack frame operations
    - Array structure
    - Structures, data alignment
    - Memory corruption
- Optimization
    - Explaining assembly code: reverse engineering
    - Data Flow graphs
    - Improving code to speed up program
- Memory
    - Locality analysis
    - Direct mapped caches
    - Associative caches
    - Hit/miss analysis

Stride in matrix multiplication

Matrix multiplication A = B x C where all the matrices ar n x n

a[i][j] = sum of (b[I][*] * c[*][j]

For n = 2

c11 = a11*b11 + a12*b21
c12 = a11*b12 + a12*b22
c21 = a21*b11 + a22*b21
c22 = a21*b12 + a22*b22

For the purposes of this analysis, we make the following assumptions:

Each array is an n × n array of double, with sizeof(double) == 8.

There is a single cache with a 32-byte block size (B = 32).

The array size n is so large that a single matrix row does not fit in the L1 cache.

The compiler stores local variables in registers, and thus references to local variables inside loops do not require any load or store instructions.

Stride in matrix multiplication

(a) Version ijk

```
for (i = 0; i < n; I++)
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += A[i][k]*B[k][j];
   C[i][j] += sum;
  }
```

Look at cache misses per inner loop.

Since block size = 32 and word size = 8, stride 1 gives miss rate = 0.25, according to

miss rate = min (1, (wordsize × k)/B)

Stride n yields miss rate 1 because n is very large.

Program versions

(a) Version ijk

```
for (i = 0; i < n; i++)
  for (j = 0; j < n; j++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += A[i][k]*B[k][j];
   C[i][j] += sum;
  }
```

(b) Version jik

```
for (j = 0; j < n; j++)
  for (i = 0; i < n; i++) {
    sum = 0.0;
    for (k = 0; k < n; k++)
      sum += A[i][k]*B[k][j];
   C[i][j] += sum;
  }
```

(c) Version jki

```
for( j = 0; j < n; j++)
  for (k = 0; k < n; k++) {
    r = B[k][j];
    for (i = 0; i < n; i++)
      C[i][j] += A[i][k]*r ;
 }
```

(d) Version kji

```
for( k = 0; k < n; k++)
  for (j = 0; j < n; j++) {
    r = B[k][j];
    for (i = 0; i < n; i++)
      C[i][j] += A[i][k]*r ;
 }
```

(e) Version kij

```
for( k = 0; k < n; k++)
  for (i = 0; i < n; i++) {
    r = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += r*B[k][j] ;
 }
```

(f) Version ikj

```
for( i = 0 i < n; i++)
  for (k = 0; k < n; k++) {
    r = A[i][k];
    for (j = 0; j < n; j++)
      C[i][j] += r*B[k][j] ;
 }
```

Miss rates

Stride/miss rate for each array
in inner loop.

Version	A	B	C	total	memory accesses
a	1/0.25	n/1	0	1.25	2
b	1/0.25	n/1	0	1.25	2
c	n/1	0	n/1	2	3
d	n/1	0	n/1	2	3
e	0	1/0.25	1/0.25	0.5	3
f	0	1/0.25	1/0.25	0.5	3

Shows 3 categories, two version have same total rate.

Versions a,b have less memory accesses but the store to c[i][j] in c,d will have a write hit all the time. Miss rate is a better predictor of speed than the number of memory accesses.

Versions e,f are the fastest with a miss rate ¼ times the slowest. Need to expand the analysis to include the middle loop.

Locality Examples

Consider the following array of structures:

```
#define N 1000
typedef struct {
  double vel[2];
  int acc[2];
  } point;

point p[N];
```

Size of one element in p is: 2 x 8 + 2 x 4 = 24

Offset to p[i].vel[j] = 24 x i + j x 8
Offset to p[i].acc[j] = 24 x i + 16 + j x 4

```
void clear1(point *p, int n)
{
int i, j;

for (i = 0; i < n; i++) {
  for (j = 0; j < 2; j++)
    p[i].vel[j] = 0;
  for (j = 0; j < 2; j++)
    p[i].acc[j] = 0;
   }
 }
```

First 10 memory accesses

i	j	Statement	offset
0	0	p[0].vel[0]	0
0	1	p[0].vel[1]	8
0	0	p[0].acc[0]	16
0	1	p[0].acc[1]	20
1	0	p[1].vel[0]	24
1	1	p[1].vel[1]	32
1	0	p[1].acc[0]	40
1	1	p[1].acc[1]	44
2	0	p[2].vel[0]	48
2	1	p[2].vel[1]	56

Memory Hierarchy

Locality Examples

```
void clear2(point *p, int n)
{
int i, j;

for (i = 0; i < n; i++) {
  for (j = 0; j < 2; j++) {
    p[i].vel[j] = 0;
    p[i].acc[j] = 0;
    }
  }
}
```

Offset to p[i].vel[j] = 24 x i + j x 8
Offset to p[i].acc[j] = 24 x i + 16 + j x 4

First 10 memory accesses

I	j	Statement	offset
0	0	p[0].vel[0]	0
0	0	p[0].acc[0]	16
0	1	p[0].vel[1]	8
0	1	p[0].acc[1]	20
1	0	p[1].vel[0]	24
1	0	p[1].acc[0]	40
1	1	p[1].vel[1]	32
1	1	p[1].acc[1]	44
2	0	p[2].vel[0]	48
2	0	p[2].acc[0]	64

Locality Examples

```
void clear3(point *p, int n)
{
int i, j;

for (j = 0; j < 2 ; j++) {
  for (i = 0; i < n; i++)
    p[i].vel[j] = 0;
  for (i = 0; i < n; i++)
    p[i].acc[j] = 0;
  }
}
```

Offset to p[i].vel[j] = 24 x i + j x 8
Offset to p[i].acc[j] = 24 x i + 16 + j x 4

First 10 memory accesses

i	j	Statement	offset
0	0	p[0].vel[0]	0
1	0	p[1].vel[0]	24
2	0	p[2].vel[0]	48
0	0	p[0].acc[0]	16
1	0	p[1].acc[0]	40
2	0	p[2].acc[0]	64
0	1	p[0].vel[1]	8
1	1	p[1].vel[1]	32
2	1	p[2].vel[1]	56
0	1	p[0].acc[1]	20

Locality Examples

What would happen if?

```
#define N 1000
typedef struct {
  double vel[3];
  int acc[3];
  } point;

point p[N];
```

Size of one element in p is: 3 x 8 + 3 x 4 + **4** = 48

Offset to p[i].vel[j] = 48 x i + j x 8
Offset to p[i].int[j] = 48 x i + 24 + j x 4

The extra 4 is to align vel[0] on an 8 byte boundary!

Cache Configuration examples

cache config

				?	?	?	?
m	C	B	E	S	t	s	b
32	1024	4	1	256	22	8	2
32	1024	8	4	32	24	5	3
32	2048	32	32	1	27	0	5

S = C/(B*E)

b = log_2B

s = log_2S

t = m-b-s

Cache Configuration examples

cache mapping

associative examples

Cache hit/miss examples

```
typedef int array[2][2];

void transpose1(array dst, array src)
  {
  int i, j;

  for (i = 0; i < 2; i++) {
    for (j = 0; j < 2; j++) {
      dst[j][i] = src[i][j];
      }
    }
  }
```

src at address 0, dst at address 16 (powers of 2 problem?)

M = 32, B = 8, C = 16, S = 2, m = 5

address overlay

t	s	b	b	b

access sequence:

	a	t	s	b	h/m
read 0	00000	0	0	0	m
write 16	10000	1	0	0	m
read 4	00100	0	0	4	m
write 24	11000	1	1	0	m
read 8	01000	0	1	0	m
write 20	10100	1	0	4	m
read 12	01100	0	1	4	m
write 28	11100	1	1	4	m

Cache hit/miss examples

Cache size 32

M = 32, B = 8, C = 32, S = 4, m = 5

address overlay

s	s	b	b	b

access sequence:

	a	t	s	b	h/m
read 0	00000	0	0	0	m
write 16	10000	0	2	0	m
read 4	00100	0	0	4	h
write 24	11000	0	3	0	m
read 8	01000	0	1	0	m
write 20	10100	0	2	4	h
read 12	01100	0	1	4	h
write 28	11100	0	3	4	h

4 misses, 8 references rate = 0.5

Cache hit/miss examples

```
struct algae_position {
  int x;
  int y;
  };

 struct algae_position grid[16][16];   // size = 256 x 8 2048 = M, m = 11
 int total_x = 0, total_y = 0;
 int i, j;

  for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
      total_x += grid[i][j].x;
      }
    }

  for (i = 0; i < 16; i++) {
    for (j = 0; j < 16; j++) {
      total_y += grid[i][j].y;
      }
    }
```

M = 2048, B = 16, C = 1024, S = 64

Address overlay

t	s	s	s	s	s	s	b	b	b	b

x access sequence:
0, 8, 16, 24, …

On x[0] (miss) , set 0 will receive x[0], y[0], x[1], y[1]
x[1] has address 8, so access 8 is a hit. Alternating hits and misses.

y access sequence:
4, 12, 20, 28, …

On y[0] (miss) , set 0 will receive x[0], y[0], x[1], y[1]
y[1] has address 12, so access 12 is a hit. Alternating hits and misses.

x reads: 256, y reads 256. ½ miss, rate = 0.5

Cache hit/miss examples

```
for (i = 0; i < 16; i++){
  for (j = 0; j < 16; j++) {
    total_x += grid[j][i].x;
    total_y += grid[j][i].y;
    }
 }
```

Access pattern: 0, 4, 64,68, 128,132 x is always a miss and y is always a hit: rate = 0.5

Twice the size: double b makes the miss rate 0.25., double s keeps the rate the same.

```
for (i = 0; i < 16; i++){
  for (j = 0; j < 16; j++) {
    total_x += grid[i][j].x;
    total_y += grid[i][j].y;
    }
 }
```

Access pattern: 0, 4, 8, 12, 16… best stride. Set holds x[0], y[0], x[1], y[1] so x[0] is a miss and the other three are a hit rate = 0.25

Twice the size: double b makes the miss rate 1/8, double s keeps the rate the same

Data Flow Graph

```
double poly(double a[], double x, int degree)  {
long int i;
double result = a[0];
double xpwr = x; /* Equals x^i at start of loop */

for (i = 1; i <= degree; i++) {
  result += a[i] * xpwr;
  xpwr = x * xpwr;
  }
  return result;
}

void main() {
double a[128]  ;
double x =  3;
int degree = 128 ;

poly( a,x,degree ) ;
printf( "%d \n", x ) ;
}
```

Data Flow Graph

poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
End of assembler dump.
```

From the code, it is clear that upon entry we have:

%xmm0	x
%rdi	addr(a)
%esi	degree

%xmm3	receives x
%xmm0	receives a[0] which is result
%rsi	receives degree
%xmm1	receives x which is xpwr
%eax	receives 1

Main compile

```
0x00000000004004f9 <+0>:    sub    $0x408,%rsp
0x0000000000400500 <+7>:    mov    %rsp,%rdi
0x0000000000400503 <+10>:   mov    $0x80,%esi
0x0000000000400508 <+15>:   movsd  0x130(%rip),%xmm0
0x0000000000400510 <+23>:   callq  0x4004c4 <poly>
0x0000000000400515 <+28>:   movsd  0x123(%rip),%xmm0
0x000000000040051d <+36>:   mov    $0x400638,%edi
0x0000000000400522 <+41>:   mov    $0x1,%eax
0x0000000000400527 <+46>:   callq  0x4003b8 <printf@plt>
0x000000000040052c <+51>:   add    $0x408,%rsp
0x0000000000400533 <+58>:   retq
```

Optimization

Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a

%xmm3	%xmm2	%xmm0	%xmm1	%rax	%rdi	%rsi

%xmm3	%xmm2	%xmm0	%xmm1	%rax	%rdi	%rsi

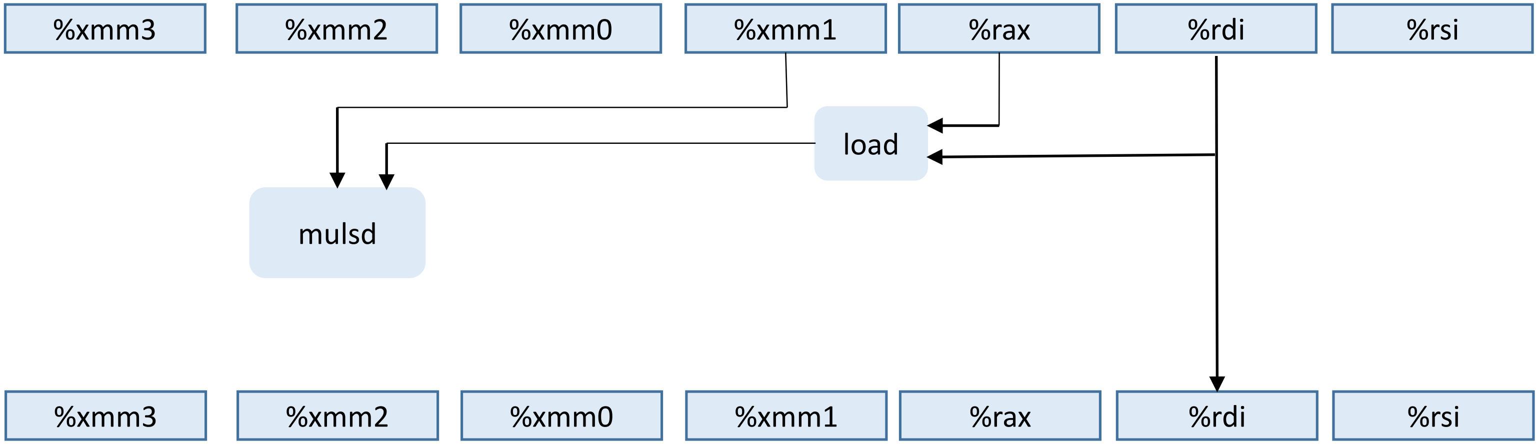
Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a



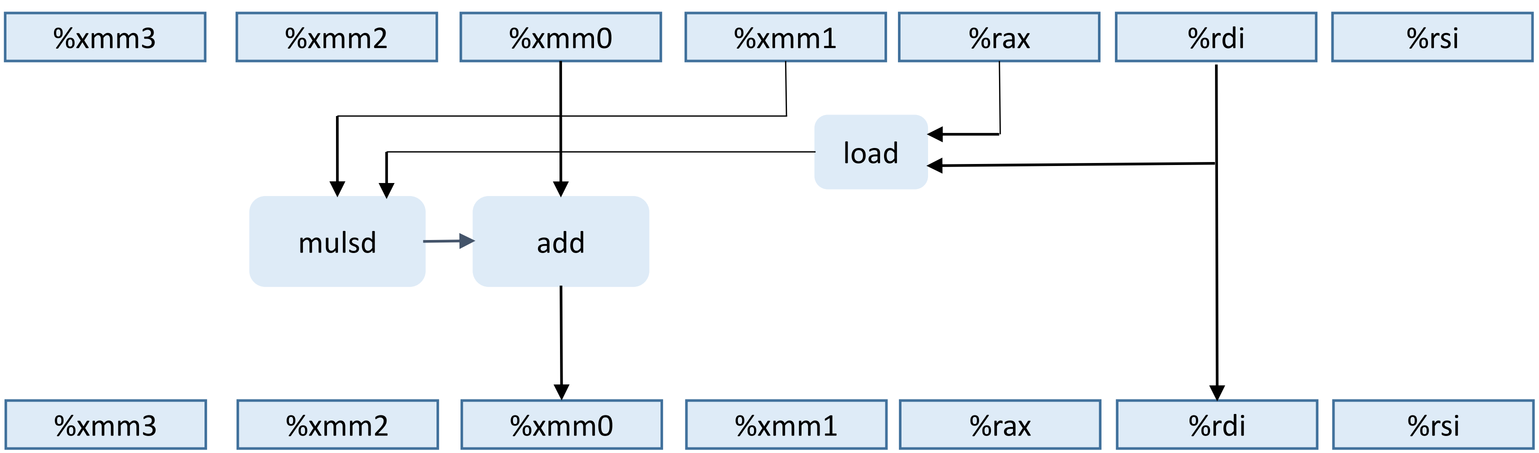
Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a



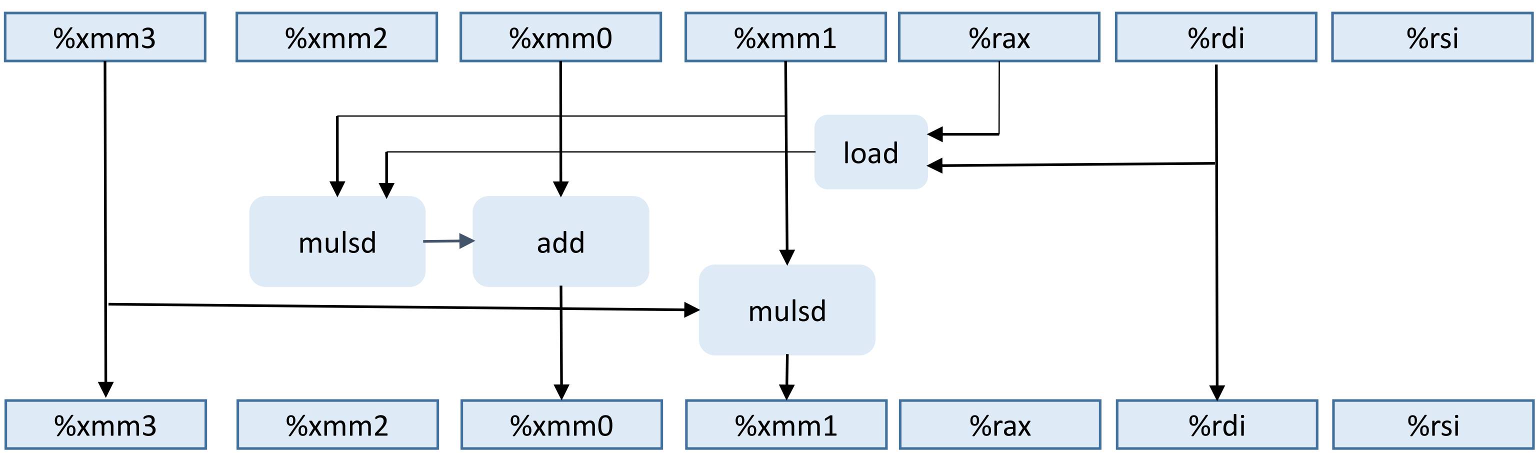
Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a



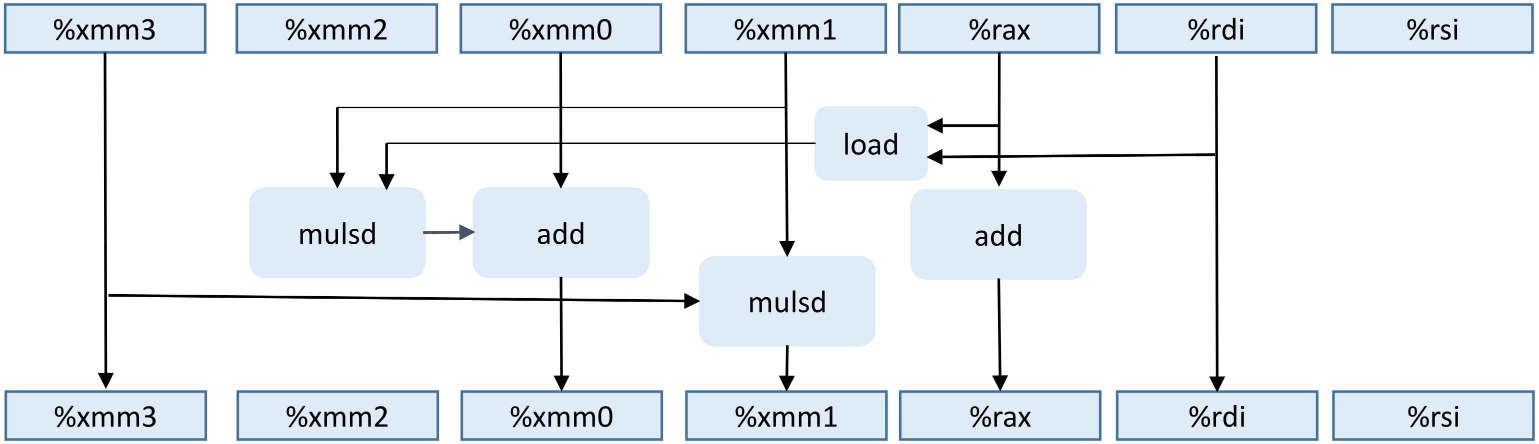
Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a



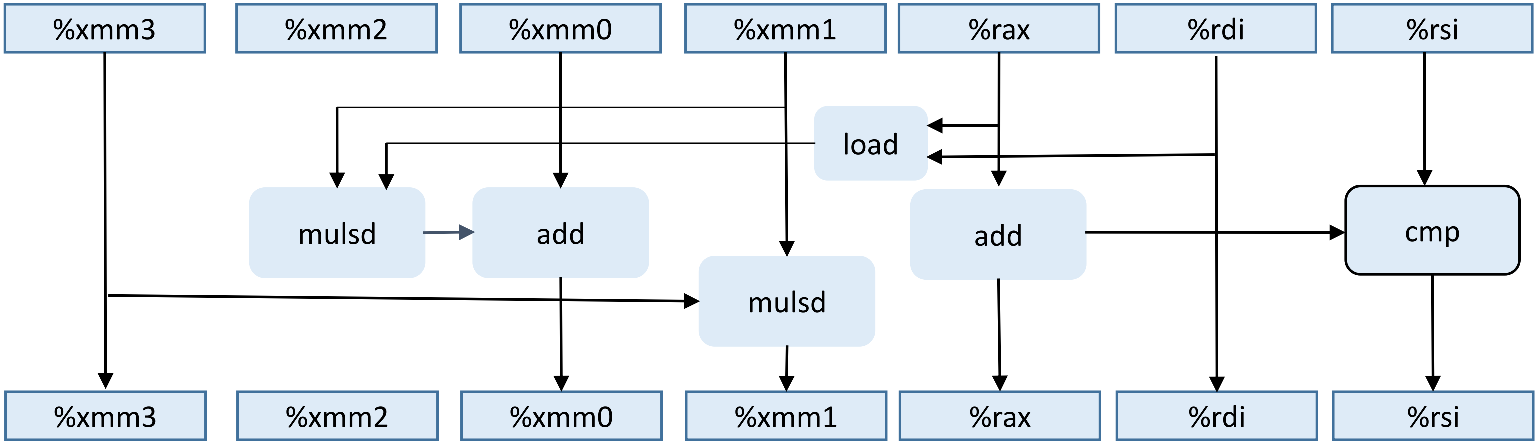
Data Flow Graph

Here is the assembly code of the poly function compiled with -O.

```
0x00000000004004c4 <+0>:    movapd %xmm0,%xmm3
0x00000000004004c8 <+4>:    movsd  (%rdi),%xmm0
0x00000000004004cc <+8>:    movslq %esi,%rsi
0x00000000004004cf <+11>:   test   %rsi,%rsi
0x00000000004004d2 <+14>:   jle    0x4004f7 <poly+51>
0x00000000004004d4 <+16>:   movapd %xmm3,%xmm1
0x00000000004004d8 <+20>:   mov    $0x1,%eax
0x00000000004004dd <+25>:   movapd %xmm1,%xmm2
0x00000000004004e1 <+29>:   mulsd  (%rdi,%rax,8),%xmm2
0x00000000004004e6 <+34>:   addsd  %xmm2,%xmm0
0x00000000004004ea <+38>:   mulsd  %xmm3,%xmm1
0x00000000004004ee <+42>:   add    $0x1,%rax
0x00000000004004f2 <+46>:   cmp    %rsi,%rax
0x00000000004004f5 <+49>:   jle    0x4004dd <poly+25>
0x00000000004004f7 <+51>:   repz retq
```

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a

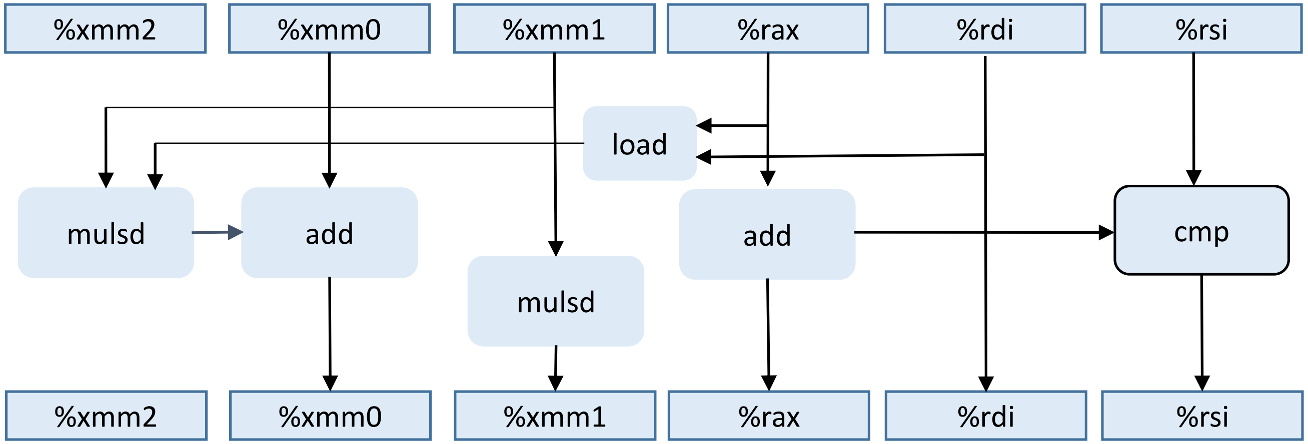


Data Flow Graph

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a

Because %xmm3 does not depend on the outcome of the loop

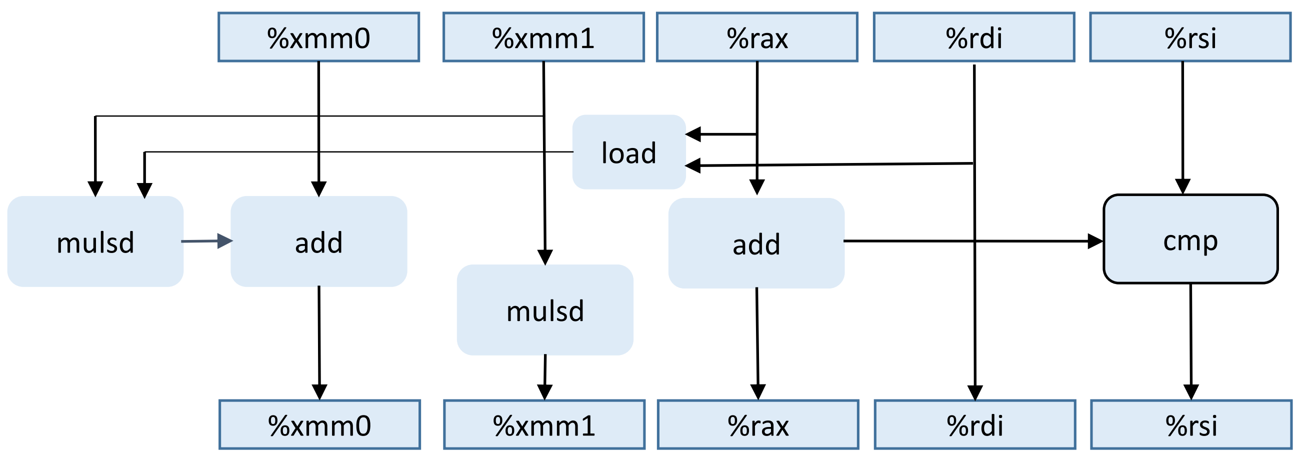


Data Flow Graph

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a

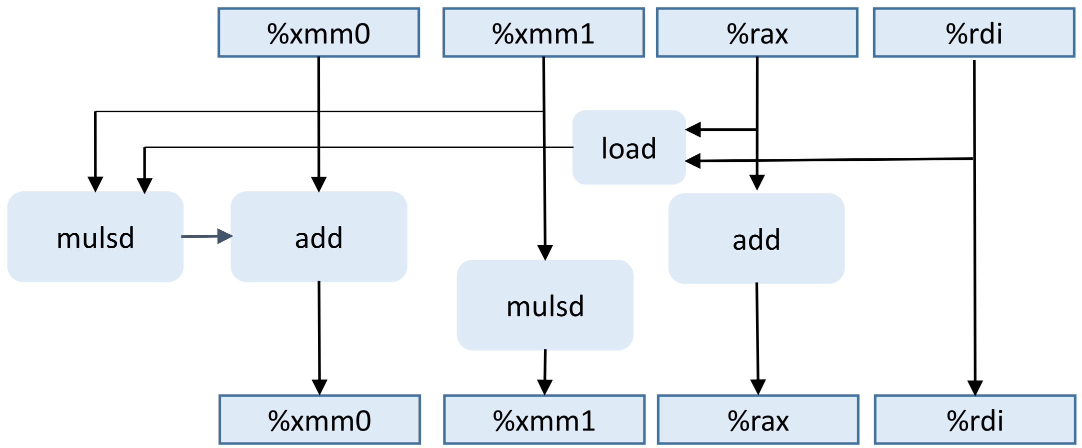
Because %xmm2 does not depend on the outcome of the loop



Data Flow Graph

We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is I
%rsi is degree
%rdi is address of a



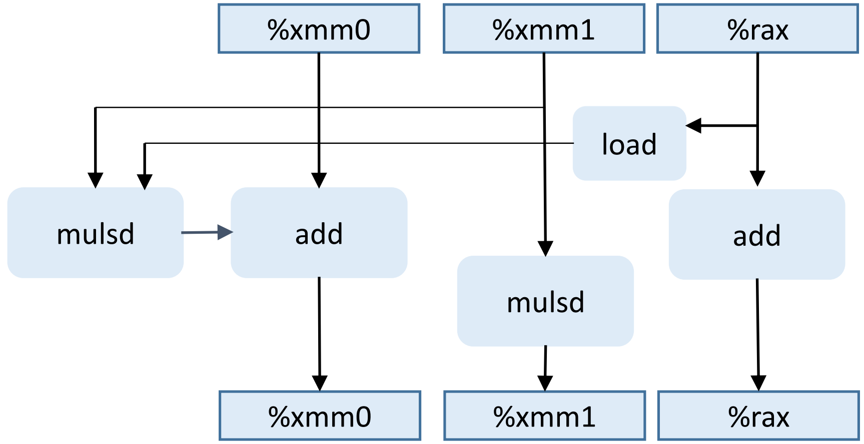
Because the compare outcome is done within the loop

Optimization

Data Flow Graph

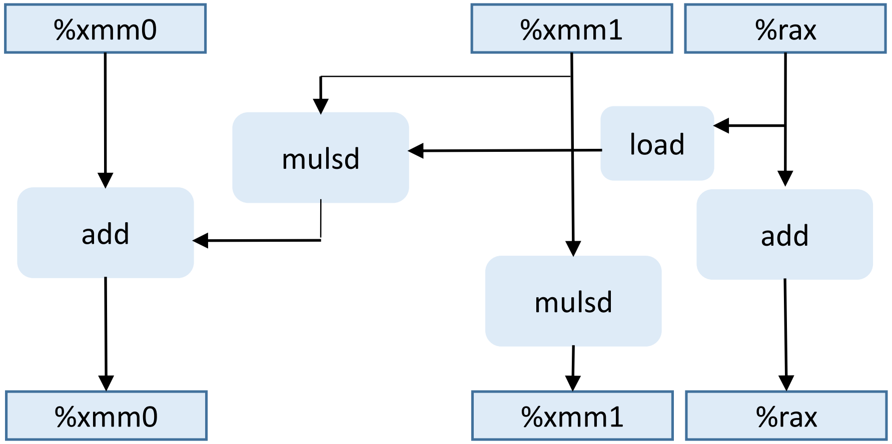
We know that:

%xmm0 is result
%xmm1 is xpwr
%xmm2 is nothing
%xmm3 is x
%rax is i
%rsi is degree
%rdi is address of a



Because %rdi is read only

Data Flow Graph



Tentative Lecture Schedule

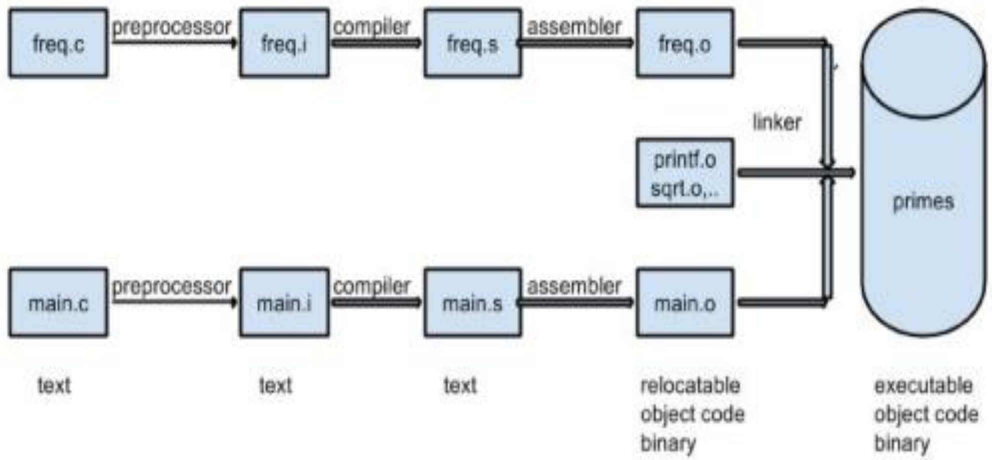
Lecture	Date	Topic
14	11/19/2014	Linking
15	11/24/2014	Exception Control Flow
16	11/26/2014	Virtual Memory
	11/30/2014	Lab 3 due
17	12/1/2014	Virtual Memory (Lab 4 issued)
18	12/3/2014	Concurrent Programming
19	12/8/2014	Concurrent Programming
20	12/10/2014	Review (Lab 4 due)
	12/16/2014	Final Exam 8AM-11AM
	11/28/2014	No Discussion - Thanksgiving Holiday
		* No office hour

Linking

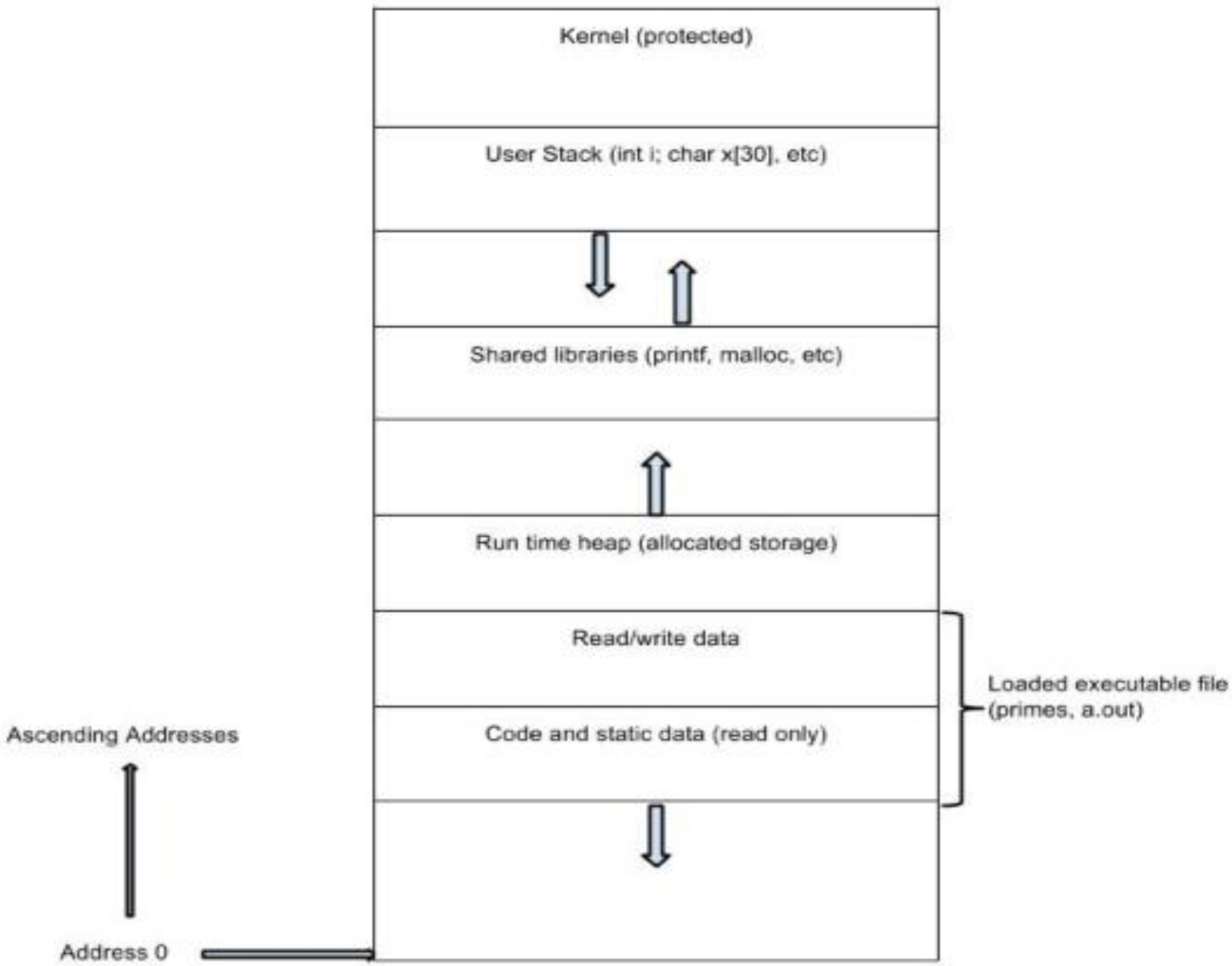
Pre-process, compile, assemble link.

Can stop/start process anywhere, combine any of these files using gcc or g++.
Pre-process, compile, assemble link.

gcc –O1 –lm –o primes freq.c main.c



Layout of program in memory



Linking

Combine/collect code – create executable

1. Combine separately compiled programs.
2. Add library routines (not shared)
3. Relocation

```
#include <stdio.h>

int a[100] ;
int b = 99 ;

int poly()
  {
  int i ;

  int sum = 0 ;
  for( i=0; i<100; i++ )
    sum = sum+a[i] ;

  return sum ;
  }

void main()
  {
  printf( "%d \n", poly() ) ;
  }
```

main()

```
0x00000000004004ff <+0>:    push   %rbp
0x0000000000400500 <+1>:    mov    %rsp,%rbp
0x0000000000400503 <+4>:    mov    $0x0,%eax
0x0000000000400508 <+9>:    callq  0x4004c4 <poly>
0x000000000040050d <+14>:   mov    %eax,%edx
0x000000000040050f <+16>:   mov    $0x400628,%eax
0x0000000000400514 <+21>:   mov    %edx,%esi
0x0000000000400516 <+23>:   mov    %rax,%rdi
0x0000000000400519 <+26>:   mov    $0x0,%eax
0x000000000040051e <+31>:   callq  0x4003b8 <printf@plt>
0x0000000000400523 <+36>:   leaveq
0x0000000000400524 <+37>:   retq
```

poly()

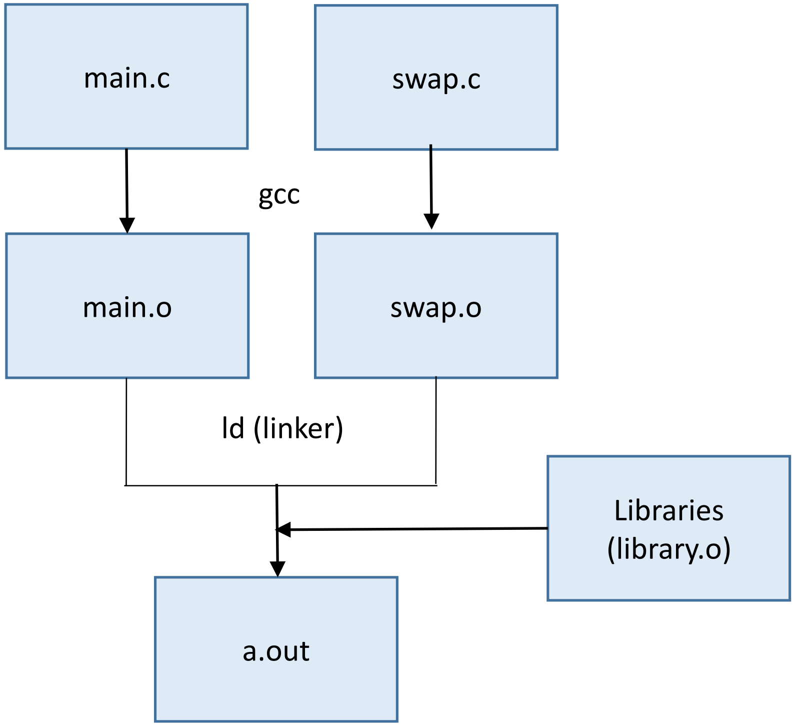
```
0x00000000004004c4 <+0>:    push   %rbp
0x00000000004004c5 <+1>:    mov    %rsp,%rbp
0x00000000004004c8 <+4>:    movl   $0x0,-0x4(%rbp)
0x00000000004004cf <+11>:   movl   $0x0,-0x8(%rbp)
0x00000000004004d6 <+18>:   jmp    0x4004eb <poly+39>
0x00000000004004d8 <+20>:   mov    -0x8(%rbp),%eax
0x00000000004004db <+23>:   cltq
0x00000000004004dd <+25>:   mov    0x600920(,%rax,4),%eax
0x00000000004004e4 <+32>:   add    %eax,-0x4(%rbp)
0x00000000004004e7 <+35>:   addl   $0x1,-0x8(%rbp)
0x00000000004004eb <+39>:   cmpl   $0x63,-0x8(%rbp)
0x00000000004004ef <+43>:   jle    0x4004d8 <poly+20>
0x00000000004004f1 <+45>:   mov    0x2003f5(%rip),%eax        # 0x6008ec <b>
0x00000000004004f7 <+51>:   add    %eax,-0x4(%rbp)
0x00000000004004fa <+54>:   mov    -0x4(%rbp),%eax
0x00000000004004fd <+57>:   leaveq
0x00000000004004fe <+58>:   retq
```

Linking

Combine/collect code – create executable

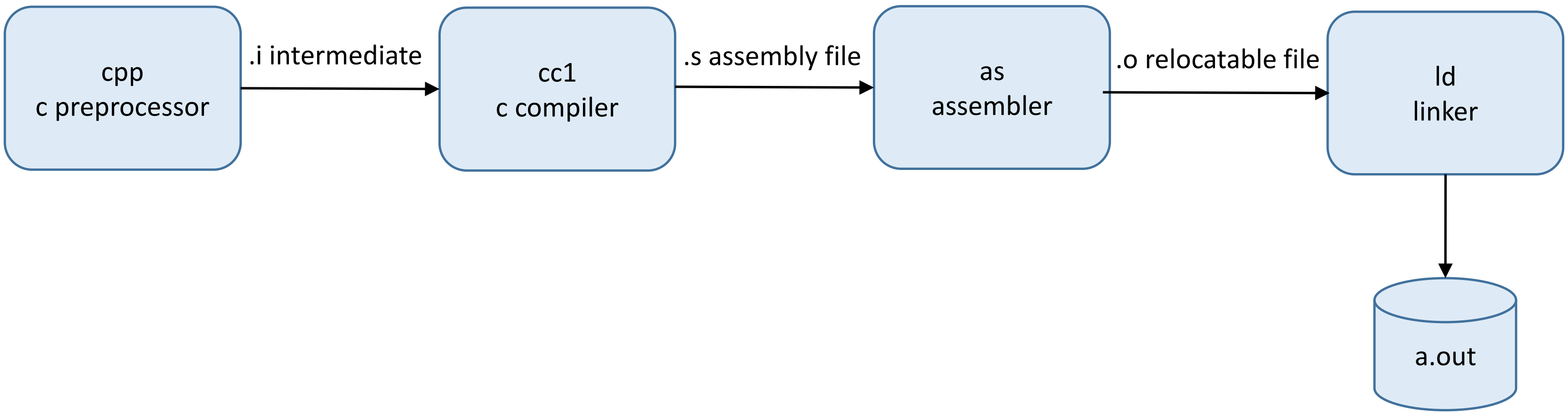
```
/* main.c */
 void swap();

 int buf[2] = {1, 2};

 int main()
 {
 swap();
 return 0;
 }


 /* swap.c */
 extern int buf[];

 int *bufp0 = &buf[0];
 int *bufp1;

 void swap()
 {
 int temp;

bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

gcc main.c swap.c library.o



Linking

gcc is actually



Static Linking

Programs contain internal and external symbols.

buf is an external symbol

Symbol resolution matches the buf in swap with the buf in main.

```
/* main.c */
void swap();

int buf[2] = {1, 2};

int main()
{
swap();
return 0;
}
```

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
int temp;

bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

Relocation

Assembler creates code relative to start of module.

Relocation changes offsets to account for multiple modules.

```
main()
 0x00000000004004ff <+0>:    push   %rbp
0x0000000000400500 <+1>:    mov    %rsp,%rbp
0x0000000000400503 <+4>:    mov    $0x0,%eax
0x0000000000400508 <+9>:    callq  0x4004c4 <poly>
0x000000000040050d <+14>:   mov    %eax,%edx
0x000000000040050f <+16>:   mov    $0x400628,%eax
0x0000000000400514 <+21>:   mov    %edx,%esi
0x0000000000400516 <+23>:   mov    %rax,%rdi
0x0000000000400519 <+26>:   mov    $0x0,%eax
0x000000000040051e <+31>:   callq  0x4003b8 <printf@plt>
0x0000000000400523 <+36>:   leaveq
0x0000000000400524 <+37>:   retq
poly()
 0x00000000004004c4 <+0>:    push   %rbp
0x00000000004004c5 <+1>:    mov    %rsp,%rbp
0x00000000004004c8 <+4>:    movl   $0x0,-0x4(%rbp)
0x00000000004004cf <+11>:   movl   $0x0,-0x8(%rbp)
0x00000000004004d6 <+18>:   jmp    0x4004eb <poly+39>
0x00000000004004d8 <+20>:   mov    -0x8(%rbp),%eax
0x00000000004004db <+23>:   cltq
0x00000000004004dd <+25>:   mov    0x600920(,%rax,4),%eax
0x00000000004004e4 <+32>:   add    %eax,-0x4(%rbp)
0x00000000004004e7 <+35>:   addl   $0x1,-0x8(%rbp)
0x00000000004004eb <+39>:   cmpl   $0x63,-0x8(%rbp)
0x00000000004004ef <+43>:   jle    0x4004d8 <poly+20>
0x00000000004004f1 <+45>:   mov    0x2003f5(%rip),%eax        # 0x6008ec <b>
0x00000000004004f7 <+51>:   add    %eax,-0x4(%rbp)
0x00000000004004fa <+54>:   mov    -0x4(%rbp),%eax
0x00000000004004fd <+57>:   leaveq
0x00000000004004fe <+58>:   retq
```

Object files

Relocatable object file (.o) can be re-linked but not executed.

Executable object file (a.out) can be executed and re-linked (with an option)

Shared object file: dynamically linked.

Relocatable object file

ELF Header
.text: machine code
.rodata: read only data
.data: initialized globals
.bss: unitialized globals (description)
.symtab: symbol table (globals and external function info)
.rel.text: relocation information for externals
.rel.data: relocation information for cross referenced data
.debug: -g symbols for gdb
.line: -g line numbers for gdb
.strtab: descriptive strings for .symtab
Section header table: which sections are in the table

```
/* swap.c */
extern int buf[];

int *bufp0 = &buf[0];
int *bufp1;

void swap()
{
int temp;          // temp local : not in .symtab

bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

.symtab contains a binding between it and .strtab called the ELF

Symbols

Three main types (table constructed by the compiler and sent to the assembler):

Global symbols within a module

Global symbols outside of a module

Local symbols

examp.c

```
extern int z ;
float y ;
int f() {

  int x = 0 ;
  return x ;
}
int g() {

  int x = 1 ;
  return x ;
```

y, f() and g() are global but defined in the module

z is global but defined elsewhere

Both x's are local. For gdb, they get unique names.

ELF descriptors

```
/* swap.c */
 extern int buf[];

 int *bufp0 = &buf[0];
 int *bufp1;

 void swap()
 {
 int temp;          // temp local : not in .symtab

bufp1 = &buf[1];
temp = *bufp0;
*bufp0 = *bufp1;
*bufp1 = temp;
}
```

symbol	.symtab?	type	Where	Section
buf	Yes	external	main.o	.data
bufp0	Yes	global	swap.o	.data
bufp1	Yes	global	swap.o	.bss
swap	Yes	global	swap.o	.text
temp	No			

Symbol resolution

Match up identically named external (global) symbols. Some defined within the module, some outside.

Strong and weak symbols.

Functions and initialized globals are strong.

Unitialized globals are weak (do not actualy take up space but are described).

When the linker encounters two symbols which are the same:

1. Both strong is an error (multiply defined)
2. One strong, one weak, use the strong (the weak references the strong)
3. Both weak, choose either

```
void f(void) ;
int x = 555 ;
int main() {
f() ;
}
int x ;
void f() {
x = 666 ;   // refers to the global x
}
```

Symbol resolution

```
void f(void) ;
int x = 555 ;
int main() {
f() ;
}
double x ;
void f() {
x = 6.66 ;   // refers to the global x but not check that the types match
}
```

Another example

```
void f(void) ;
int  buf[2] = { 1,2 };
int main() {
f() ;
}
Int buf[] ;
void f() {
buf[5] = 20 ;   // bad news
}
```

Static libraries

Combine pre-compiled routines into a module from a library. Called static.

printf, rand, exp2, etc

Create a library of common routines.

gcc main.c /usr/lib/libc.o

But the linker only includes those routines which are referenced in main.c. Could create a chain of references.

Static libraries ae created using the Unix archive function.

Relocation

Merging all the modules in a link step:

```
int x = 5, y = 6, z = 7 ;
void f1() {
.
.
.}
int a = 5, b = 6, c = 7 ;
void f2() {
.
.
}
```

.data section (initialized globals) becomes something like:

a, b, c, x, y, z

When above is compiled, a and x have offset 0 within f2 and f1. After combining, differenmt offsets.

Same is true for the .rss section (uninitialized globals)

When combined with main(), f1 and f2 will have offsets relative to the module.

Relocation Tables

Contained in the relocatable object module

Each entry has the offset within the module of a relocatable object and a type:

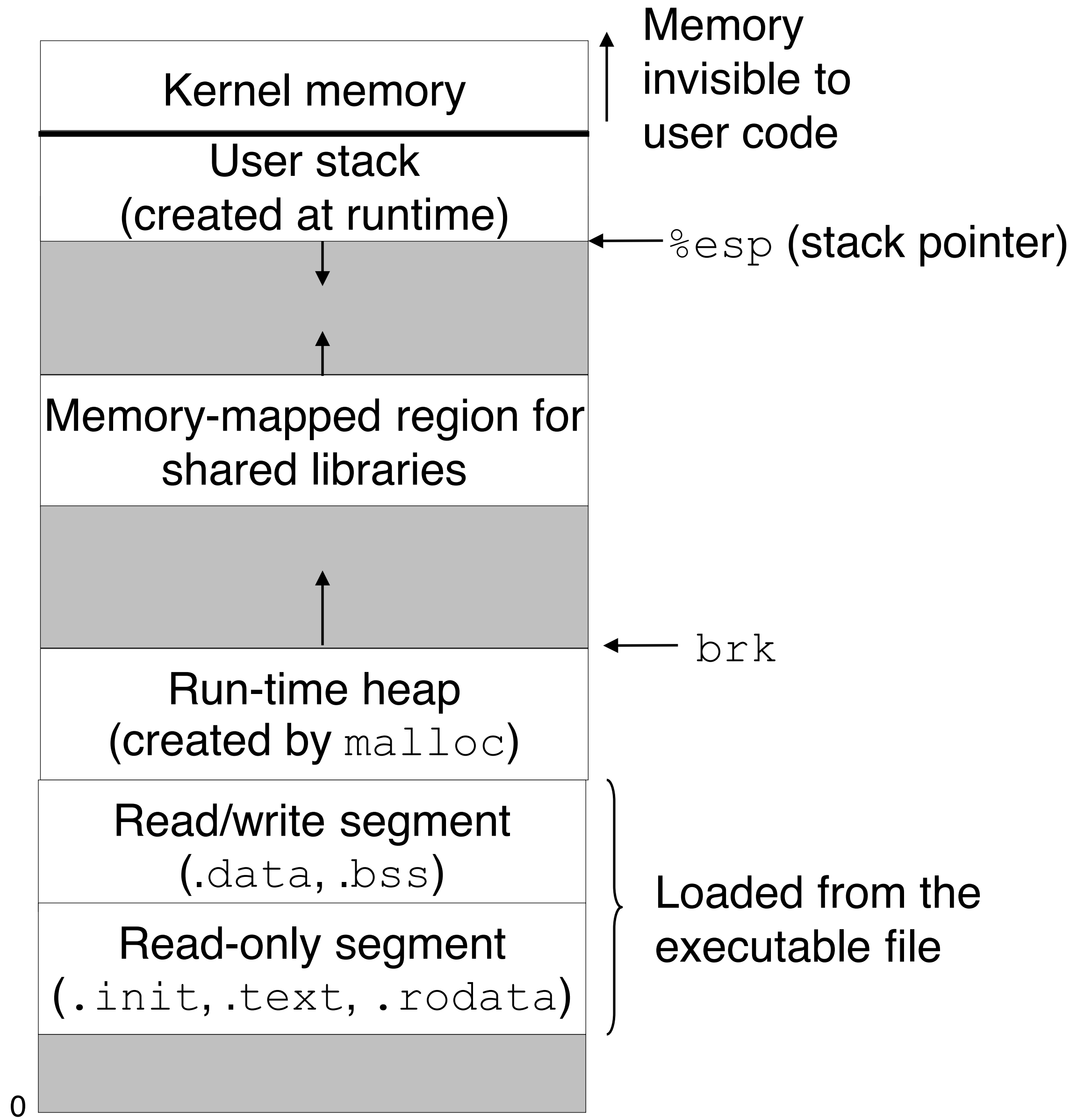
2 types are important:

1. relative to the program counter
2. absolute

Depends on how the module was compiled.

Loading

Linux memory map



Dynamic Linking

Use library routines not in the executable

Enhances the maintainablility.

Loads the library into a shared area. Provides tools to locate specific functions within the library.

Functions run in the shared library area.

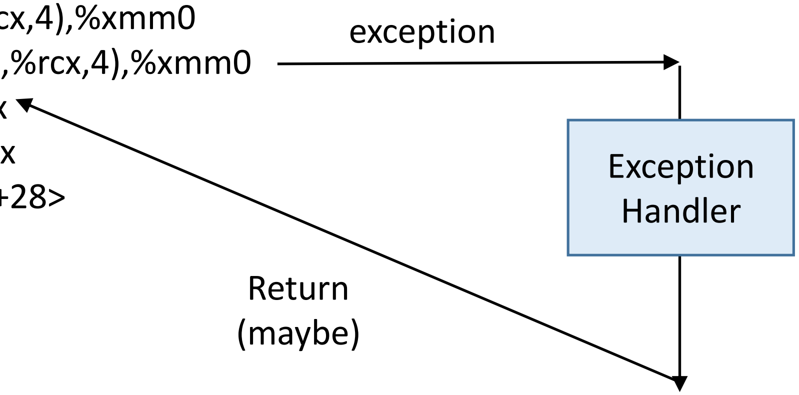
Functions which contain position independent code can be loaded without relocation. Use a “global offset table” to reference global variables.

Exceptional Control Flow

The key to concurrent execution

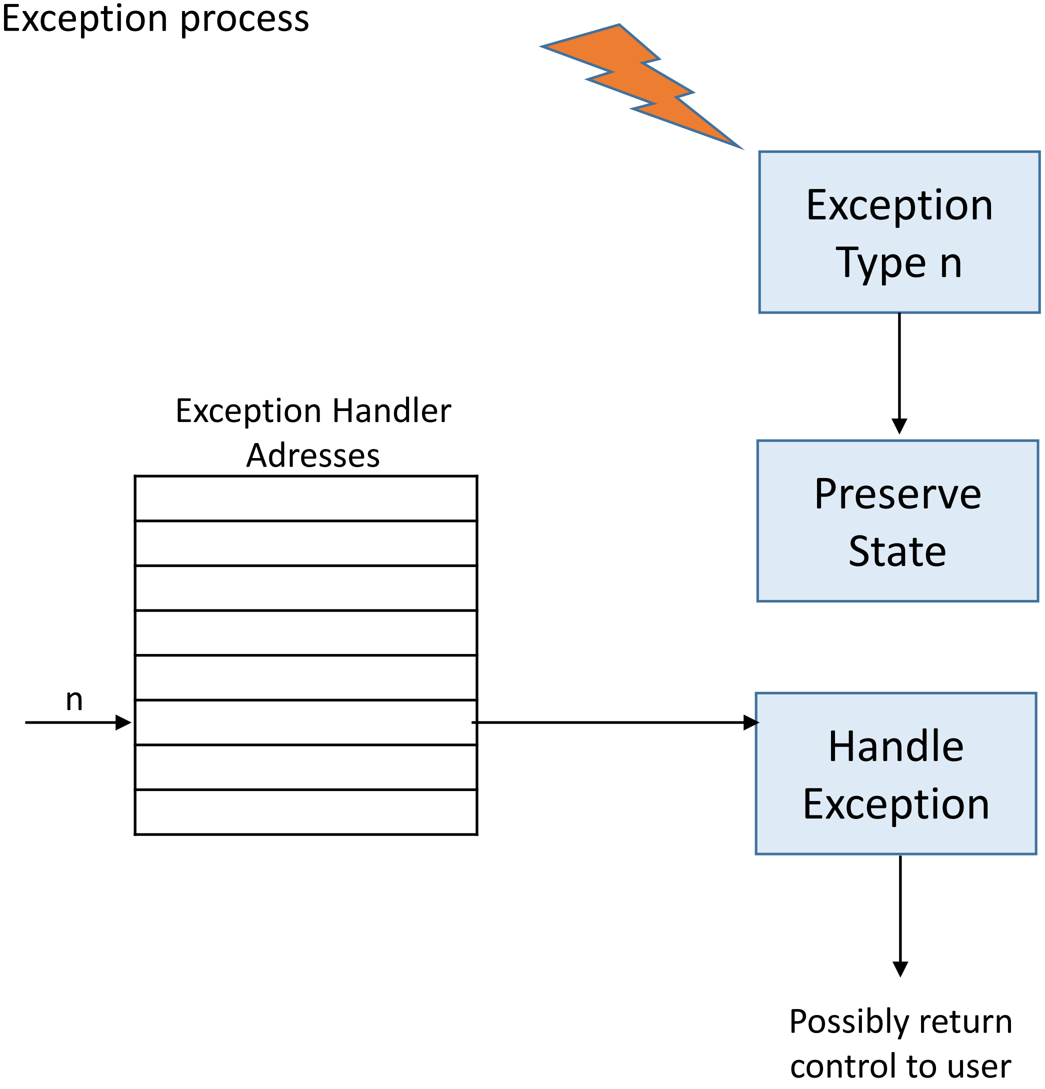
Normal execution controlled by the program counter (next instruction or jumps)

```
400b27 <+28>:   movslq %eax,%rcx
400b2a <+31>:   mulss  (%rdx,%rcx,4),%xmm0
400b2f <+36>:   mulss  0x4(%rdx,%rcx,4),%xmm0
400b35 <+42>:   add    $0x2,%eax
400b38 <+45>:   cmp    %edi,%eax
400b3a <+47>:   jl     0x400b27 <+28>
```



Exceptional Control Flow

Exception process



Classess of exceptions

Class	Source	Synchronous?	Return?
Interrupt	external	no	Always return to next instruction
Trap	self generated	yes	Always return to next instruction
Fault	Self generated, possibly recoverable	yes	Maybe return to current instruction
Abort	Not recoverable	Yes	Never return

Types of exceptions

Exception	Class
Divide Error	Fault
Protection Fault	Fault
Page Fault	Fauilt
Machine Check	Abort
OS-Defined	Interrupt or Trap
System Call	System Call

System Calls

System Call	Name
Exit	Exit
Fork	Fork
Read file	Read
Write file	Write
Open file	Open
Close file	Close
Wait for child	Waitpid
Load and run	Execve
Go to file offset	Lseek
Get process id	Getpid

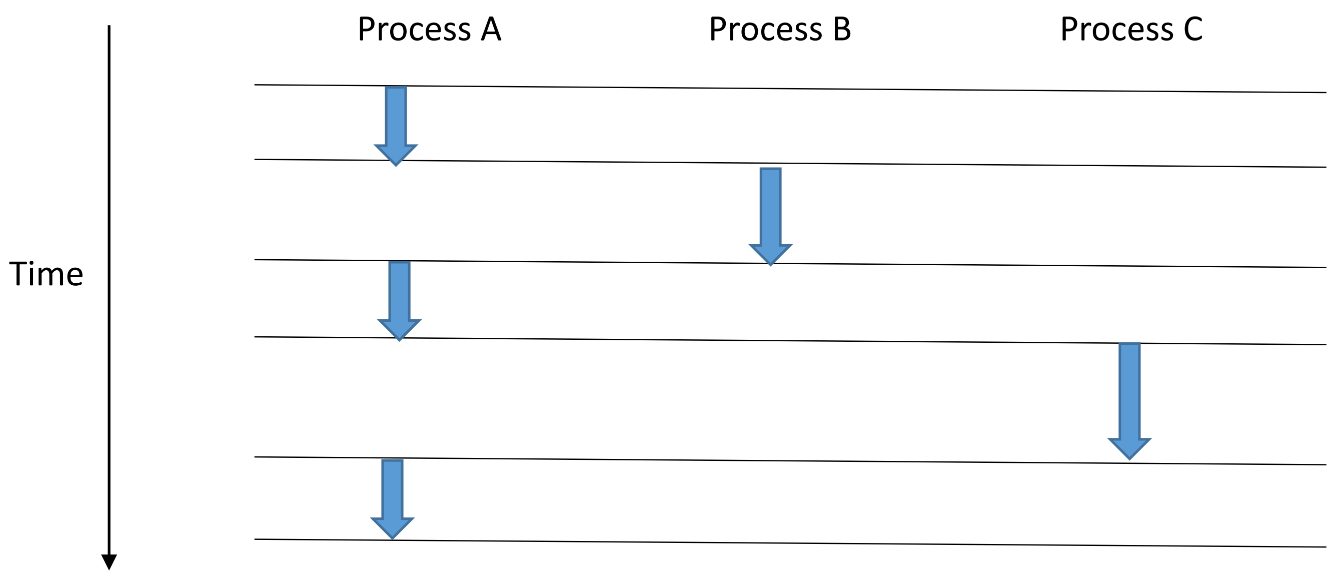
Assembly language instruction is

```
mov $n, %eax  // system call number
int   $0x80
```

Processes

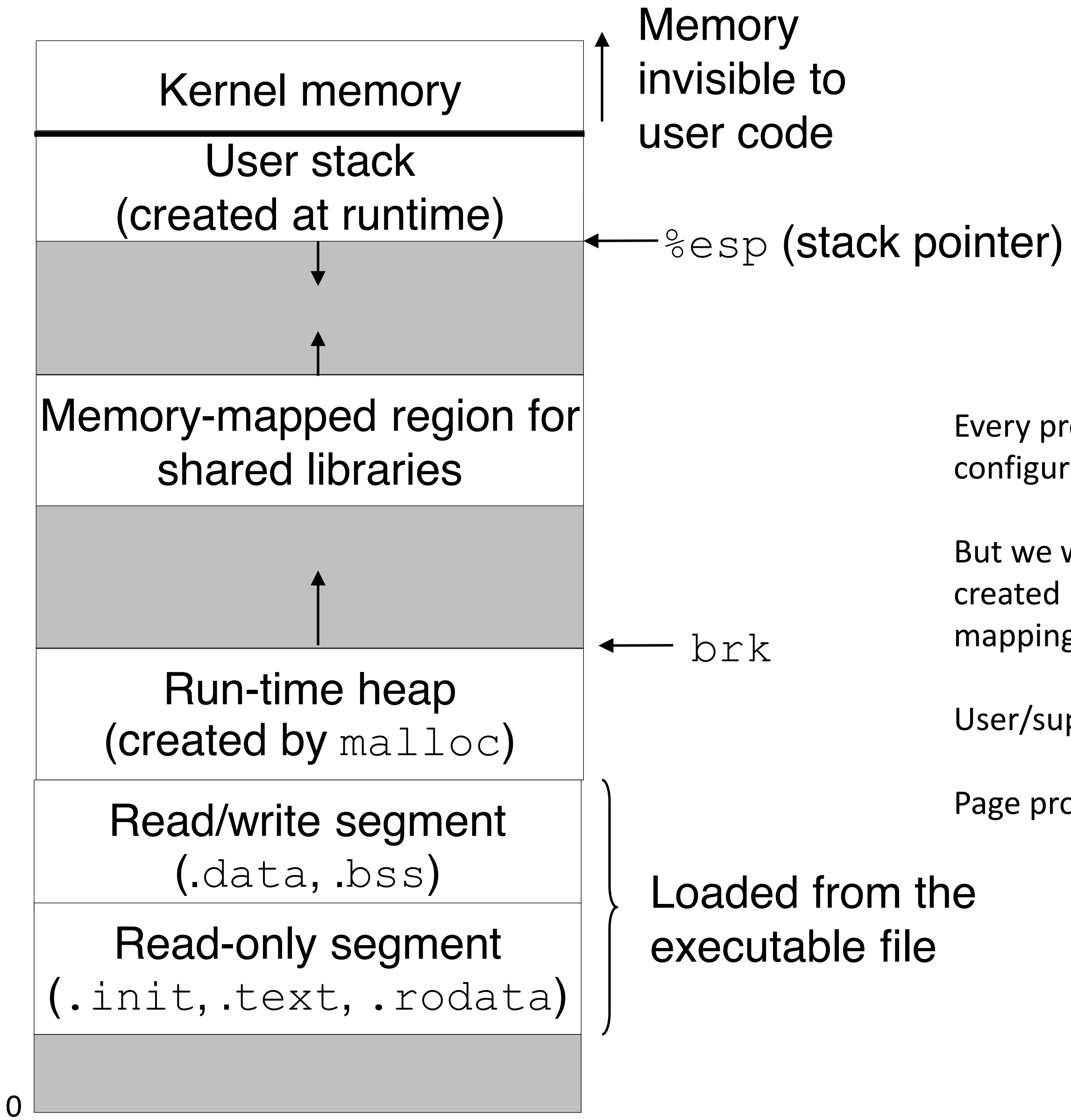
Process: an instance of program execution? Seems to be the only user of the entire resource.

Independent logical control flow.



Concurrency A:B A:C nor B:C

Linux memory map



Every process sees this configuration.

But we will see this is an illusion created by virtual memory mapping.

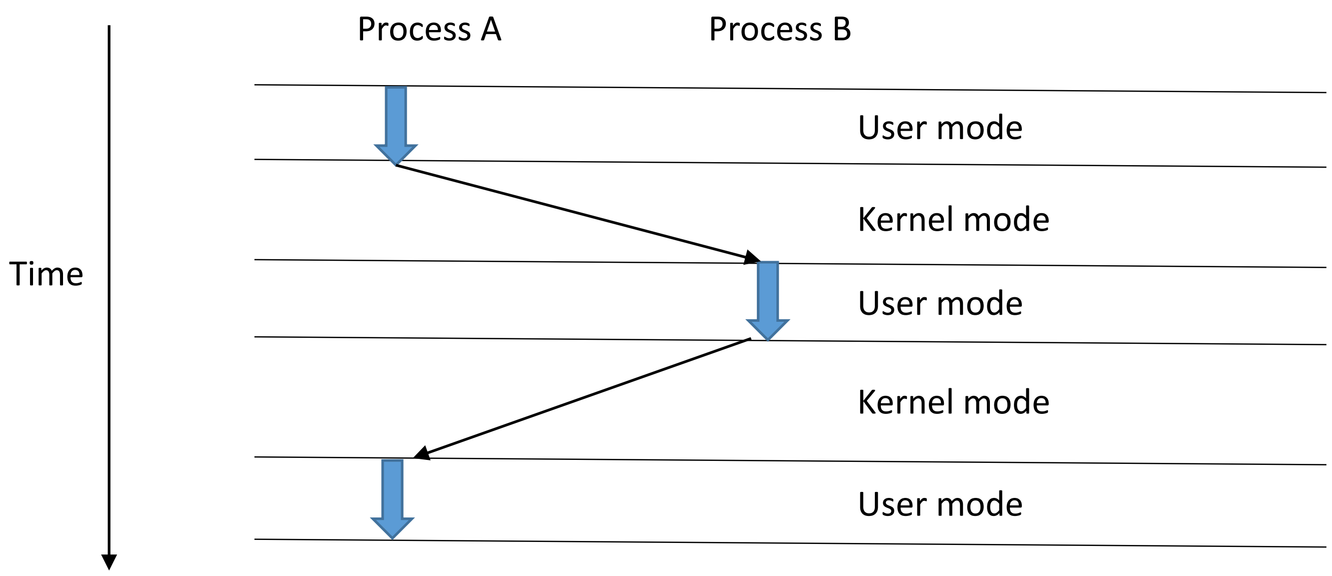
User/supervisor mode

Page protections

Context Switch

Context: registers, program counter, user stack, files.

Kernel has a scheduler, processes have priority



Process Control

Unique PID for each process

getpid: get own process ID

getppid: get parent's process ID

Processes are either

running (could be waiting)
stopped (stopped by a STOP signal, must be restarted)
terminated

exit(int status) terminates with return code

Start new process with fork() ; duplicates the conext, creates new PID.

Fork example

```
#include "csapp.h"

int main()
{
pid_t pid;
int x = 1;

pid = fork();
if (pid == 0) { /* Child */
printf("child : x=%d\n", ++x);
exit(0);
}

/* Parent */
printf("parent: x=%d\n", --x);
exit(0);
}
```

Creates new process and returns after the calling point. But it returns twice! But, pid is zero for the child, pid of the child for the parent.

Take out the exit(0) and what happens?

Parent and child now run concurrently.

Address space is duplicated initially but are private (not shared).

Files are shared.

fork example

```
#include "csapp.h"
int main()
{
fork();
fork();
fork();
printf("hello\n");
exit(0);
}
```

Main Process
```
fork();
fork();
fork();
printf("hello\n");
exit(0);
```

Child 11
```
fork();
fork();
printf("hello\n");
exit(0);
```

Child 12
```
fork();
printf("hello\n");
exit(0);
```

Child 13
```
printf("hello\n");
exit(0);
```

Child 111
```
fork();
printf("hello\n");
exit(0);
```

Child 112
```
printf("hello\n");
exit(0);
```

Child 121
```
printf("hello\n");
exit(0);
```

Child 1111
```
printf("hello\n");
exit(0);
```

Process Termination

Process which terminate go into suspended state until reaped.

It is a “zombie” process.

waitpid(pid_t, int *status, int options)

when pid > 0 wait for that child, pid = -1 wait for any child
status = -1 when there is no child to wait for

waitpid hangs the calling process but can add test options (eg NOHANG)

run fork.c

sleep and pause

run fork1.c

Run a program

execve loads and runs a program in the current context.

execve does not return: replaces the calling program.

Using execve in conjunction with fork is sort a “shell” behavior.

Example:

```
int main( int argc, char **argv ) {

execve( “program”, argv, NULL ) ;
printf( “hello\n” ) ;
}
```

hello will never come out. Must do:

```
int main( int argc, char **argv ) {

If( fork() == 0 )
  execve( “program”, argv, NULL ) ;
printf( “hello\n” ) ;
}
```

Signals

High level software implements messaging process. Many types, each has and ID: e.g. kill = ID 9.

Type of interrupt indicating an event has occurred. Similar to exceptions except signal events are handled in user mode.

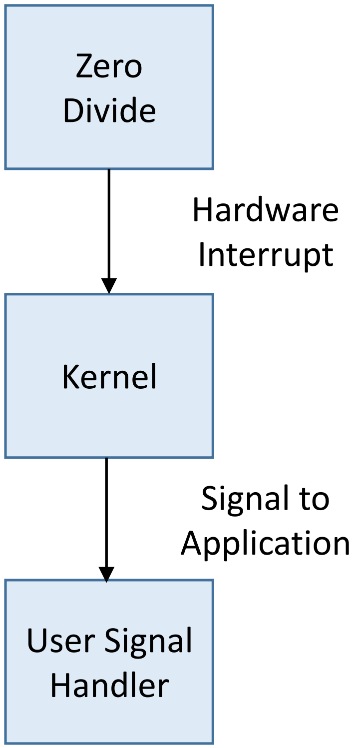
Requires implementation of signal handlers.

Signal normally sent by kernel but can be sent by a user process to itself.

Signal received by a process by being interrupted and control passing to the signal handler. Process can ignore signals.

Ignored signals are discarded.

Pending signals are queued to a level of 1.



Exceptional Control Flow

Some examples

# Name	Default action	Corresponding event
1 SIGHUP	Terminate	Terminal line hangup
2 SIGINT	Terminate	Interrupt from keyboard
3 SIGQUIT	Terminate	Quit from keyboard
4 SIGILL	Terminate	Illegal instruction
5 SIGTRAP	Terminate and dump core Trace	trap
6 SIGABRT	Terminate and dump core	Abort signal from abort function
7 SIGBUS	Terminate	Bus error
8 SIGFPE	Terminate and dump core	Floating point exception
9 SIGKILL	Terminate	Kill program
10 SIGUSR1	Terminate	User-defined signal 1
11 SIGSEGV	Terminate and dump core	Invalid memory eference (seg fault)
12 SIGUSR2	Terminate	User-defined signal 2
13 SIGPIPE	Terminate	Wrote to a pipe with no reader
14 SIGALRM	Terminate	Timer signal from alarm function
15 SIGTERM	Terminate	Software termination signal
16 SIGSTKFLT	Terminate	Stack fault on coprocessor
17 SIGCHLD	Ignore	A child process has stopped or terminated
18 SIGCONT	Ignore	Continue process if stopped
19 SIGSTOP	Stop until next SIGCONT (2)	Stop signal not from terminal
20 SIGTST	Stop until next SIGCONT	Stop signal from terminal
21 SIGTTIN	Stop until next SIGCONT	Background process read from terminal
22 SIGTTOU	Stop until next SIGCONT	Background process wrote to terminal

Process groups/handlers

A grouping of processes using an ID. Child belongs to parent group by default.

Can change process group ID using setpgid(pid, pgid) ;

Can send the same signal to every process in a group. e.g. can kill every process in a group or a single process.

signal(sig, func) function “installs” or denotes a func as the handler for signal # sig.

signal can be used to IGNore, use default (DFL) or the named function as the handler.

Example of handler

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

void handler(int sig) /* SIGINT handler */
  {
  printf("Caught SIGINT\n");
  exit(0);
  }

int main()
  {
  /* Install the SIGINT handler */
  if (signal(SIGINT, handler) == SIG_ERR)
    unix_error("signal error");

  pause(); /* Wait for the receipt of a signal */

  exit(0);
  }
```

Pending signals

Blocked signals: same signals are blocked during signal handling. 2cd signal processed after first.

Signal queue: size 1, additional signals are ignored/discarded.

Certain system calls, when interrupted by a signal are not resumed.

Run handler1

Blocking signals

Signal set contains the signals currently blocked: array of signal numbers

sigemptyset:	unblocks all signals
sigfillset:	blocks all signals
sigaddset:	adds a signal to the set
sigdelset	deletes a signal
sigprogmask:	manipulates the set (blocks/unblocks)
sigismember	tests

Synchronization

Cannot predict when child or parent runs. Different results.

Run handler3 & 4

Virtual Memory

Overview

Illusion of memory.

Actually a version of caching.

Cache line

b[0]	b[1]	…					…	b[n-2]	b[n-1]

tt…ttss…ssbb…bb is the address is physical memory. ss….ss is the line in the cache, bb…bb is the block number in the line. tt…tt validates the cache line.

Virtual address is address in “pretend” memory. Physical address is the address in main memory.

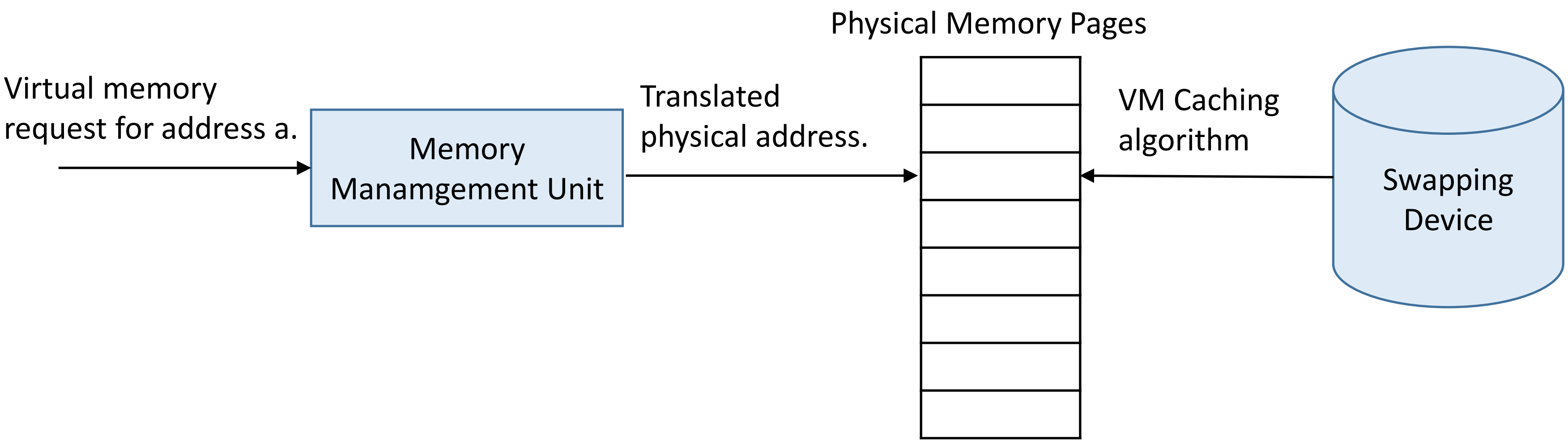
Virtual memory divided into pages. Stored in physical memory using swapping scheme.

Address translation

Virtual Memory

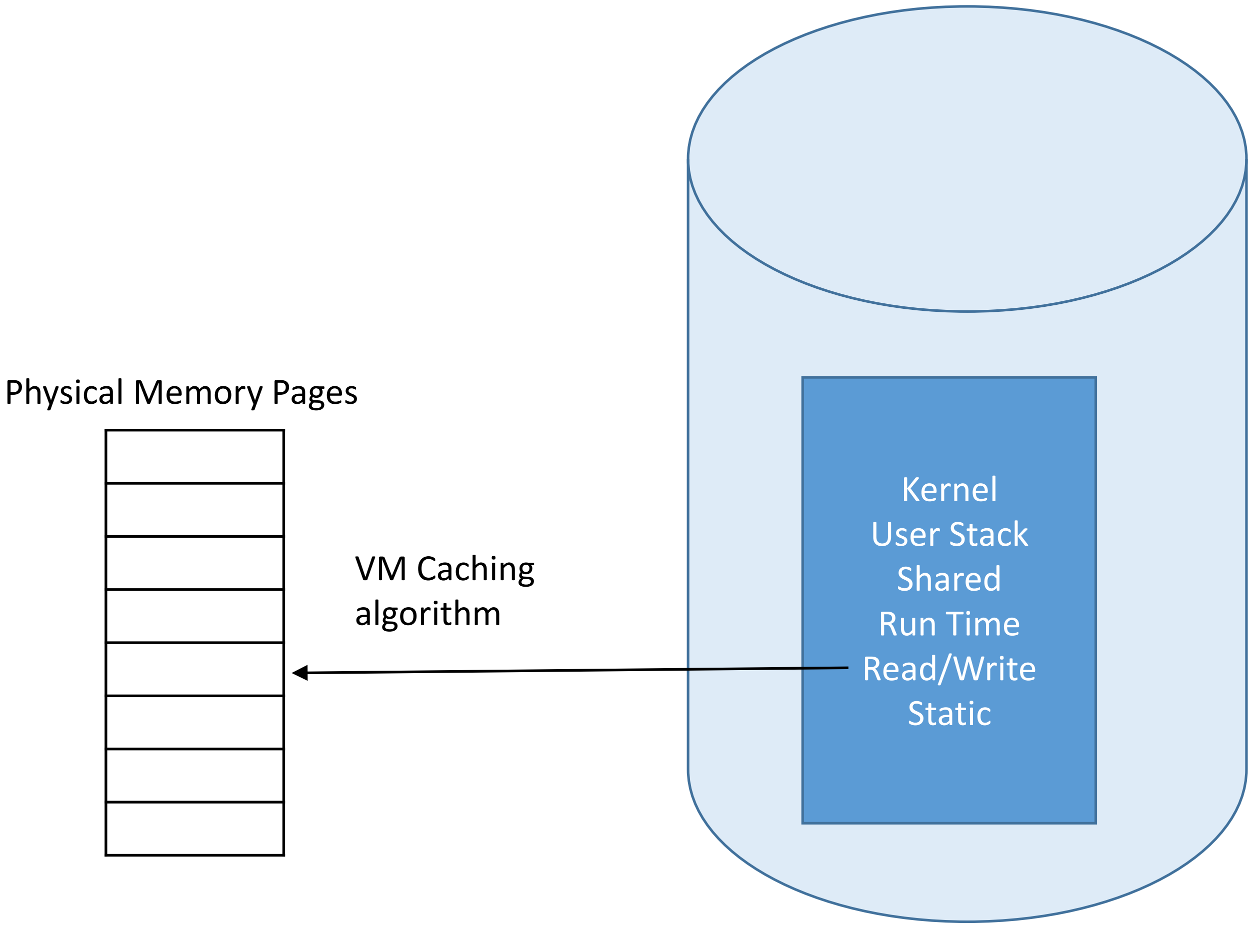
Address translation

Data fecthing



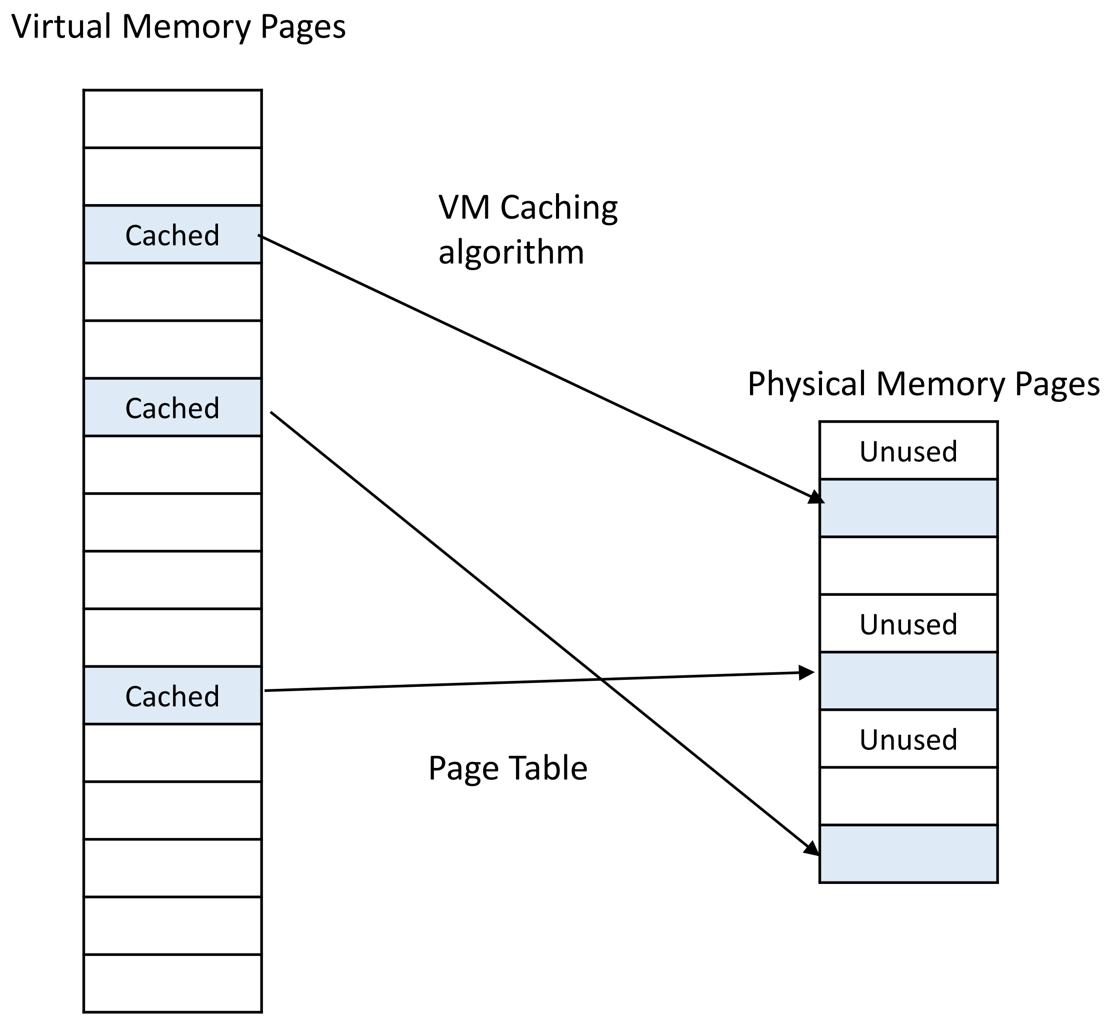
Virutal Memory

Physical appearance



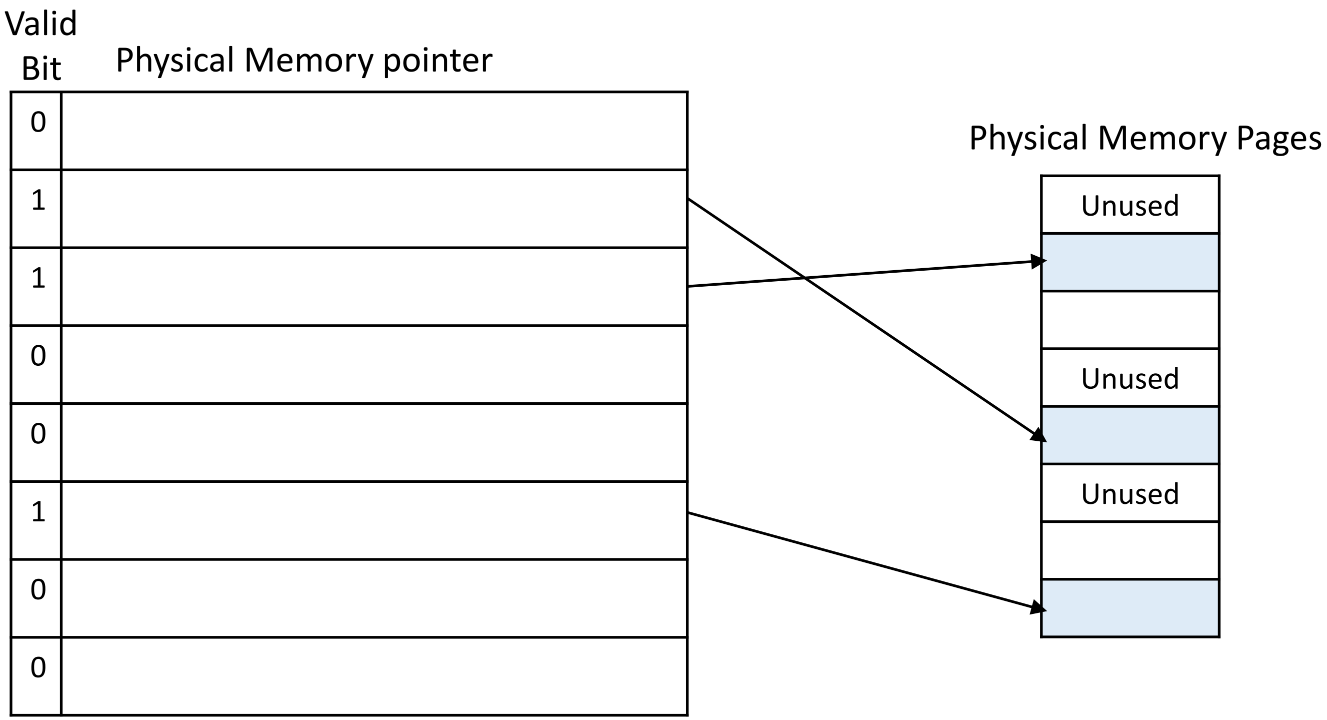
Virutal Memory

VM Bookkeeping



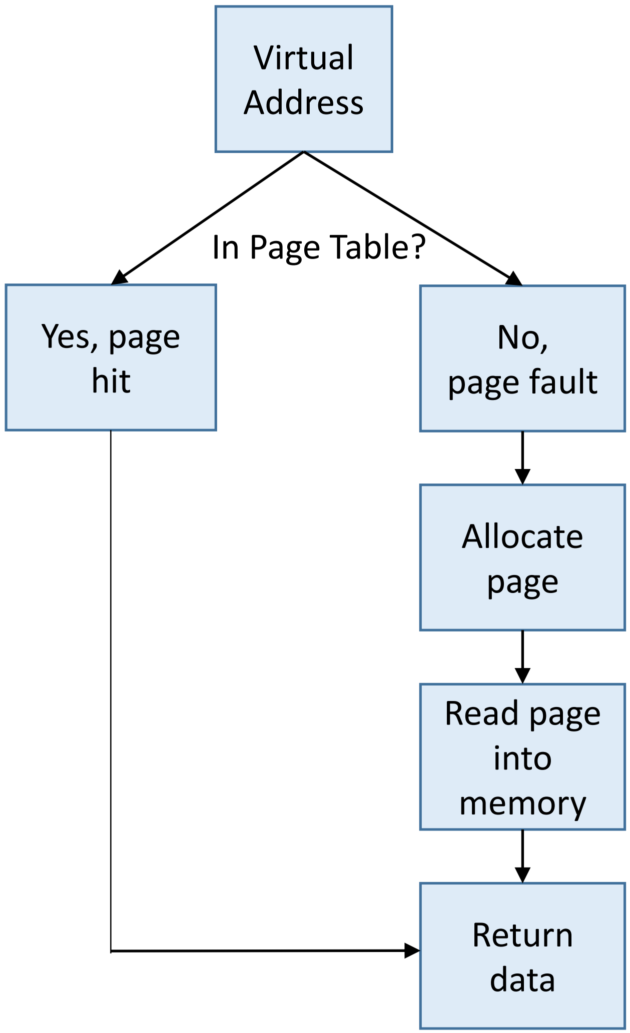
Virutal Memory

Page Table



Virutal Memory

Address request

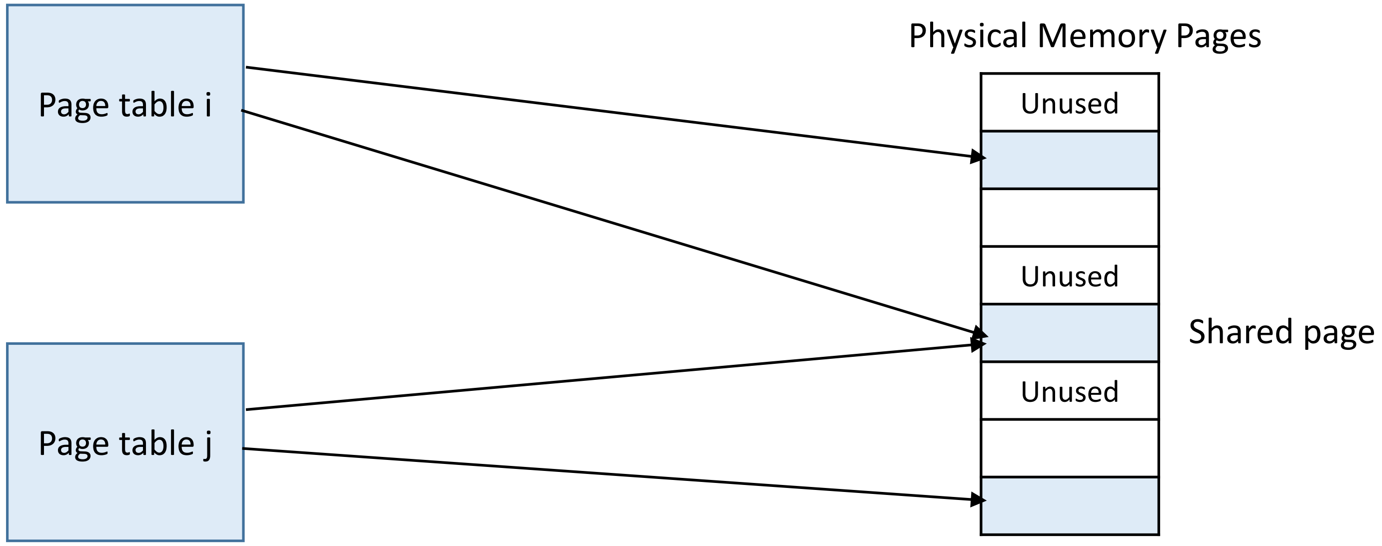


Multiple processes

Page table for each process.

Switch process, switch page table register.

Possibly shared pages (two table point to the same physical page)



Memory protection

Add protection bits to page table:

Supervisor mode, read?, write?, exectute?

Valid Bit	Protection Bits				Physical Memory pointer
0					
1					
1					
0					
0					
1					
0					
0					

Virtual Memory

Address translation terminology

Similar to caching?

Memory sizes

N = 2^n	addresses in virtual memory
M = 2^m	addresses in physical memory
P = 2^p	page size

Virtual address made up of

VPN	virtual page number
VPO	virtual page offset
TLBI	translation lookaside buffer index
TLBO	translation lookaside buffer offset

Physical address made up of:

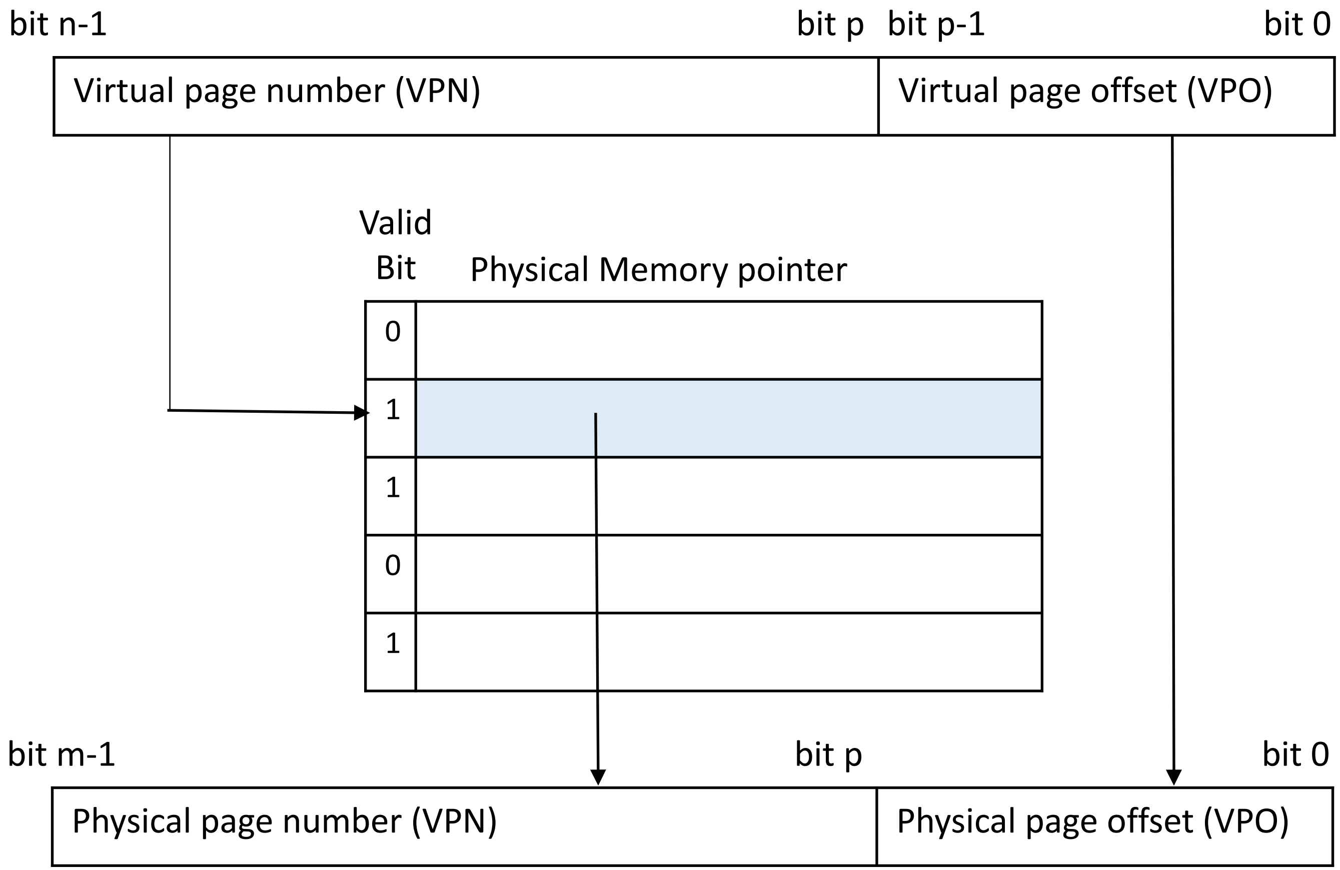
PPN	physical page number	
PPO	physical page offset	
CI	cache index	for L1 cache
CO	cache offset	
CT	cache tag	

Divde address (n bits) into sections by p

Construct mapping of virtual addresses into physical addresses (VM caching algorithm)

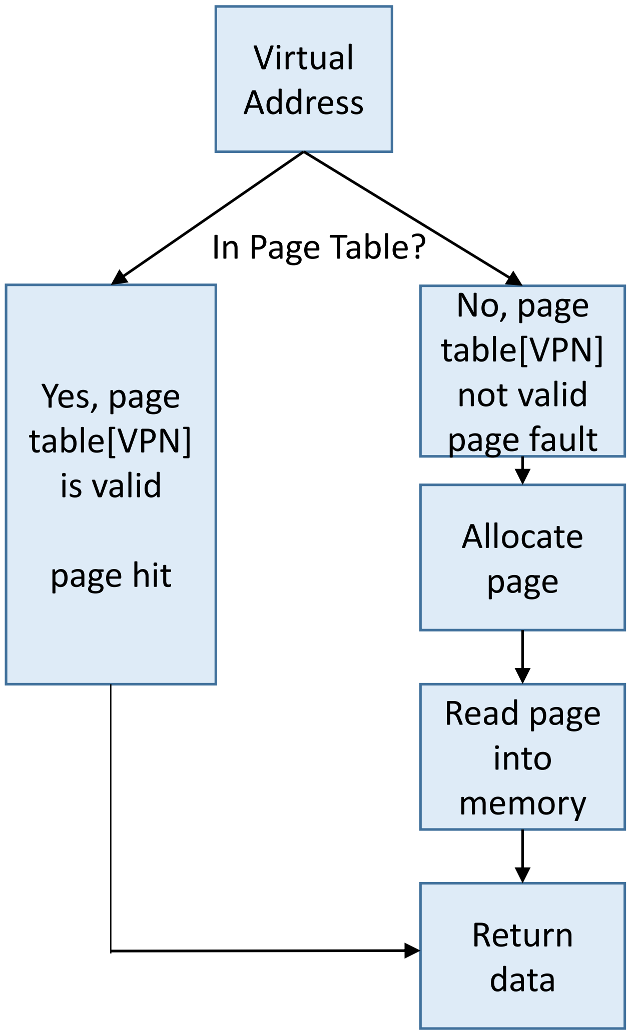
Virtual Memory

Address translation methodology



Situation for one process:
each has its own page table

Direct mapped cache?



Address translation terminology

N/M virtual pages per physical page.

Number of entries in page table? N/P. Can be quite large and resides in memory!

To reduce memory accesses to the PTE, use a cache in the memory management unit:

Translation lookaside buffer (TLB)

<table>
<tr><td>bit n-1</td><td>bit p</td><td>bit p-1</td><td>bit 0</td></tr>
<tr><td colspan="2">Virtual page number (VPN)</td><td colspan="2">Virtual page offset (VPO)</td></tr>
</table>

<table>
<tr><td>TLB Tag (TLBT)</td><td>TLB index (TLBI)</td><td>Virtual page offset (VPO)</td></tr>
<tr><td>bit n-1 … bit p+t</td><td>bit p+t-1 … bit p</td><td>bit p-1 … bit 0</td></tr>
</table>

Cache contains one block per line = page table entry for VPN, 2^t lines.

Reduces memory accesses.

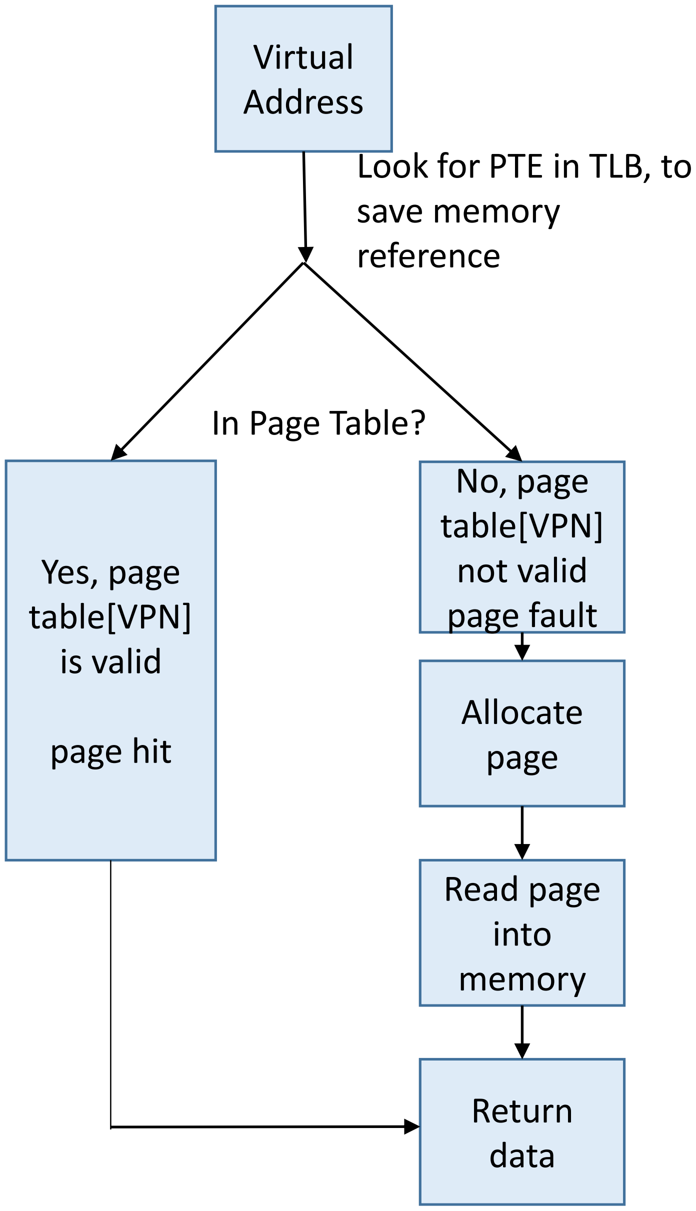
TLB diagram

TLB[0]	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry	…
TLB[1]							
⋮			⋮				
TLB[2^t-2]							
TLB[2^t-1]	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry	…

N-way associative, addressed by TLB index and TLB Tag

Virutal Memory

Direct mapped cache?



Hierarchy of page tables

Most processes use only a little bit of the memory at a time.

Most page table entries do not point to anything.

Example:

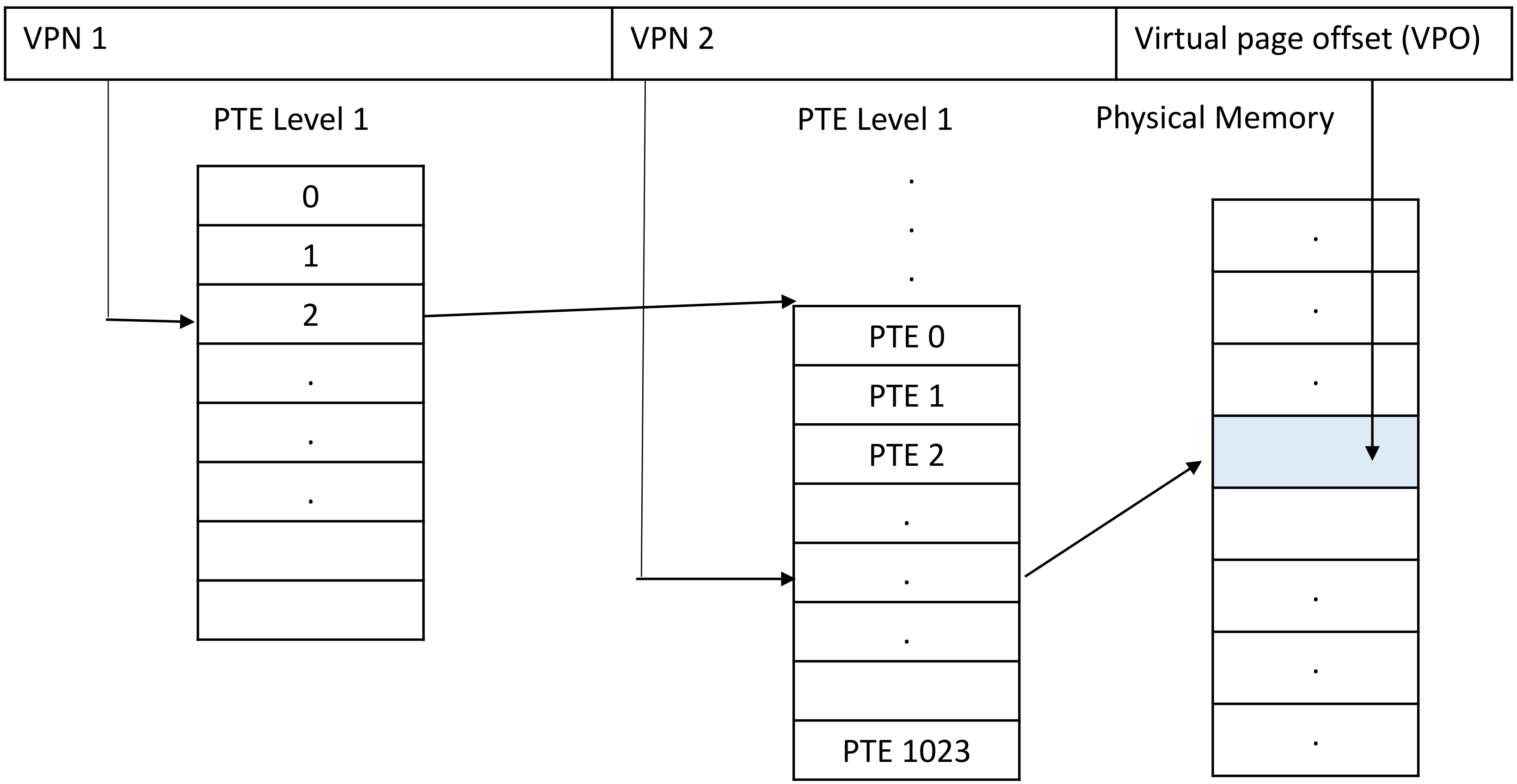
Use 32 bit address space. Divide up into 4 MB (1024, 4096 byte) sections

First level points, not to a physical address but the secondary PTE.

Second level corresponds to virtual address in its 4 MB section

Hierarchy of page tables

corrresponds to virtual address in its 4 MB section



VM Caching example

Assume m = 32 so virtual memory is 2^{32} = 4,294,967,296 bytes (4GB)
p = 12 page size is 2^{12} = 4096 bytes (4K)

Divides the memory into 2^{20} = 1,048,576 (1MB) pages

For a new process, we would allocate 1MB 4K records on the swapping disk.

Some bookkeeping mechanism to correlate page number to disk location.

In what follows, memory address go up when you go down the page.

VM Caching example

List of memory addresses on disk after load of program

	Virtual address	Disk address
read only area	0	0
	1	.
	.	.
	.	
	.	.
	x	x
read/write area	x+1	x+1
	.	NULL
	.	x+2
	y	NULL
run time heap		NULL
	.	NULL
		NULL
user stack area		NULL
	.	NULL
		NULL

Point to 4K records on executable object file (file backed memory)

Unitialized areas have no records on disk until a dirty write occurs. Afterward, they will point to 4K records in the swap file. (demand zero memory)

Address layout

Bits Virtual memory address

31	30							· · ·											12	11					· · ·						0
n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	n	o	o	o	o	o	o	o	o	o	o	o	o

n's form a 20 bit unsigned number = Virtual Page Number VPN

o's form a 12 bit unsigned number = Virtual Page Offset VPO

Virtual pages from disk will be stored in physical memory, as needed. For our example, let the physical memory be 2^{20} = 1MB or 256 4K pages

Bits Physical memory address

19			· · ·				12	11					· · ·						0
n	n	n	n	n	n	n	n	o	o	o	o	o	o	o	o	o	o	o	o

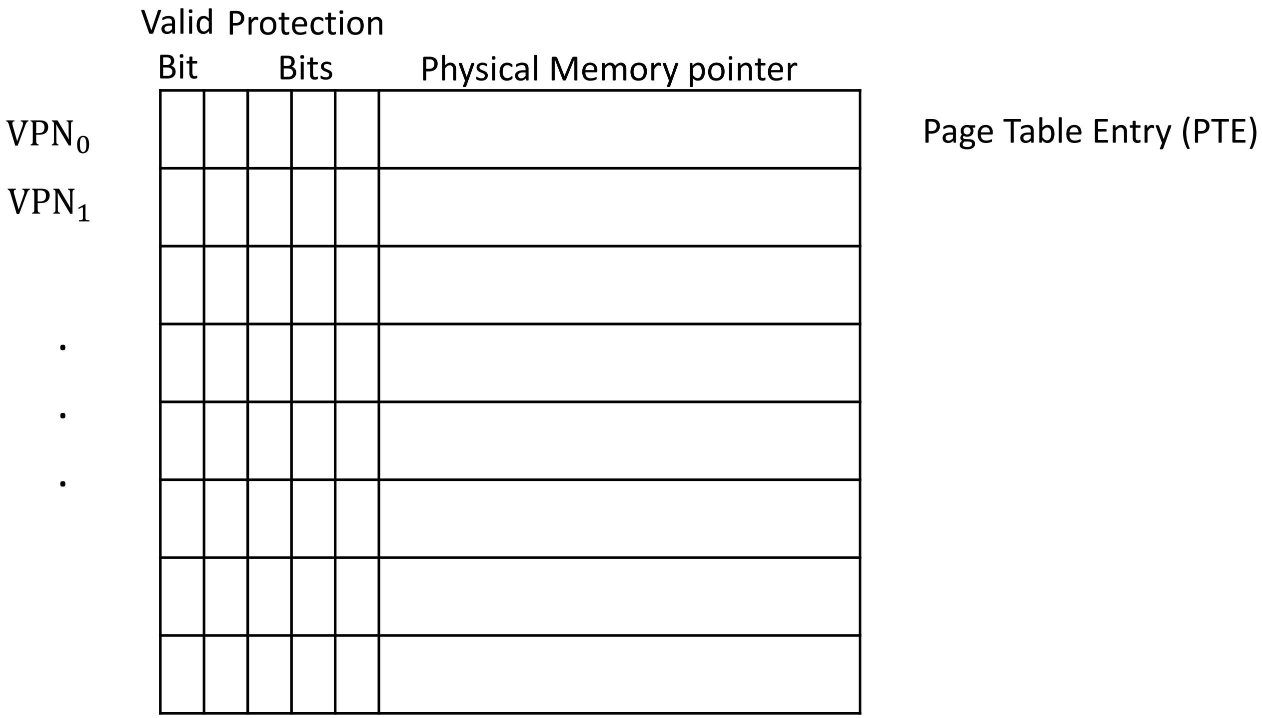
n's form a 8 bit unsigned number = Physical Page Number PPN

o's form a 12 bit unsigned number = Physical Page Offset PPO

Page Table

Disk pages (= Virtual Memory pages) will be stored in physical memory, when being used by the process.

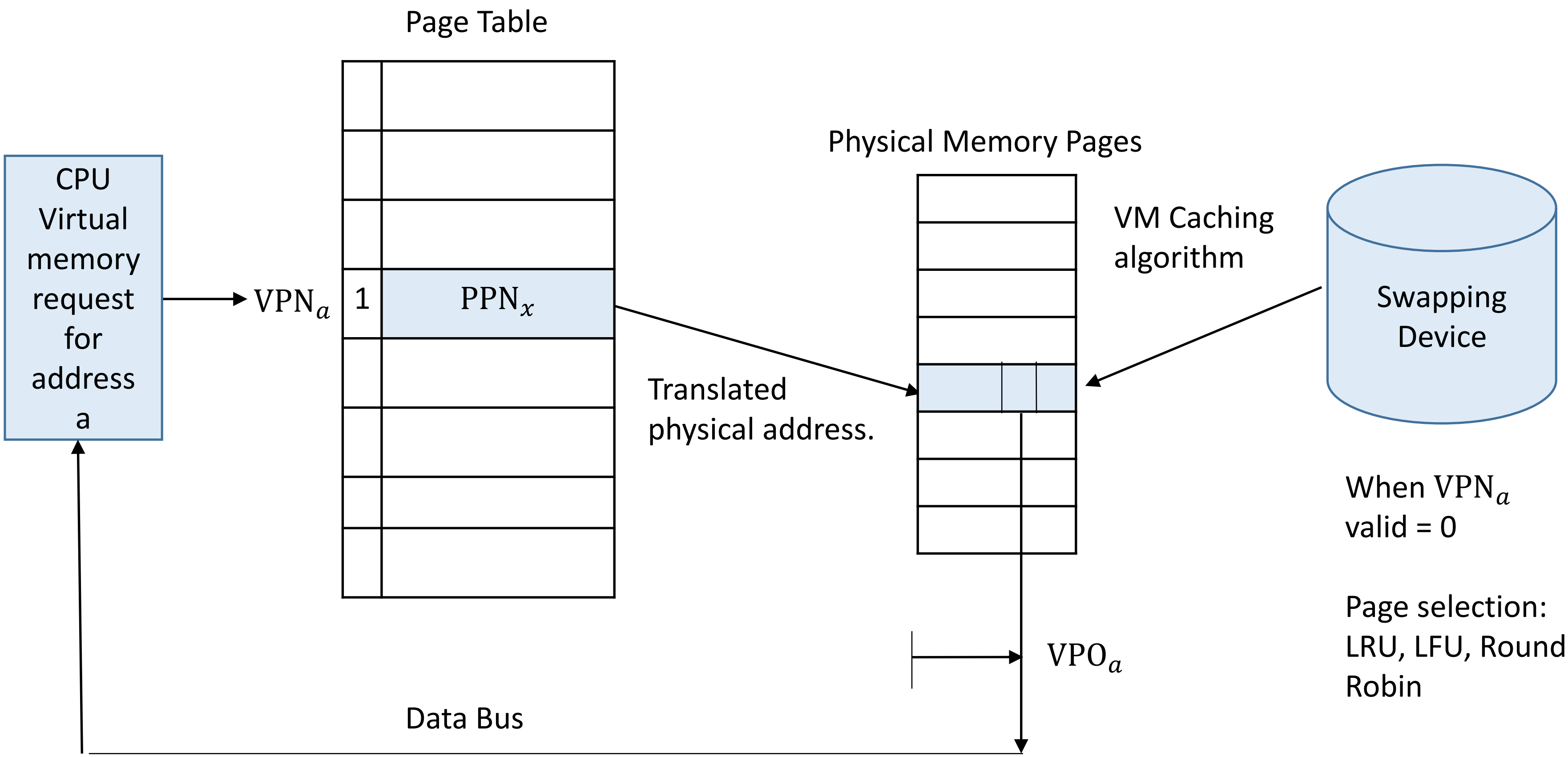
Need a table to translate VPN into PPN. 1MB entries, 4 bytes each, directly addressed by VPN.



When virtual page not in physical memory, valid bit = 0.

Virtual Memory

Address translation



When VPN_a valid = 0

Page selection: LRU, LFU, Round Robin

Translation Lookahead Buffer

Page tables are large, stored in memory. 1 entry per virtual page 1MB x 4.

Speed up with a page table entry cache: TLB. n-way associative (fast), with t lines.

Divide VPN into tag and index: for our example, assume TLB has 16 lines, 4 entries:

Bits Virtual Page Number

19			. . .					11					. . .		4	3			0
t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	t	i	i	i	i

t's form a 16 bit unsigned number = TLB Tag

i's form a 4 bit unsigned number = TLB Index

TLB diagram

TLB[0]	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry
TLB[1]												

. .

. .

. .

TLB[14]												
TLB[15]	TLB Valid	TLB Tag	PTE entry	TLB Valid	TLB Tag	PTE entry						

4-way associative, addressed by TLB index and TLB Tag

Virtual Memory

TLB example

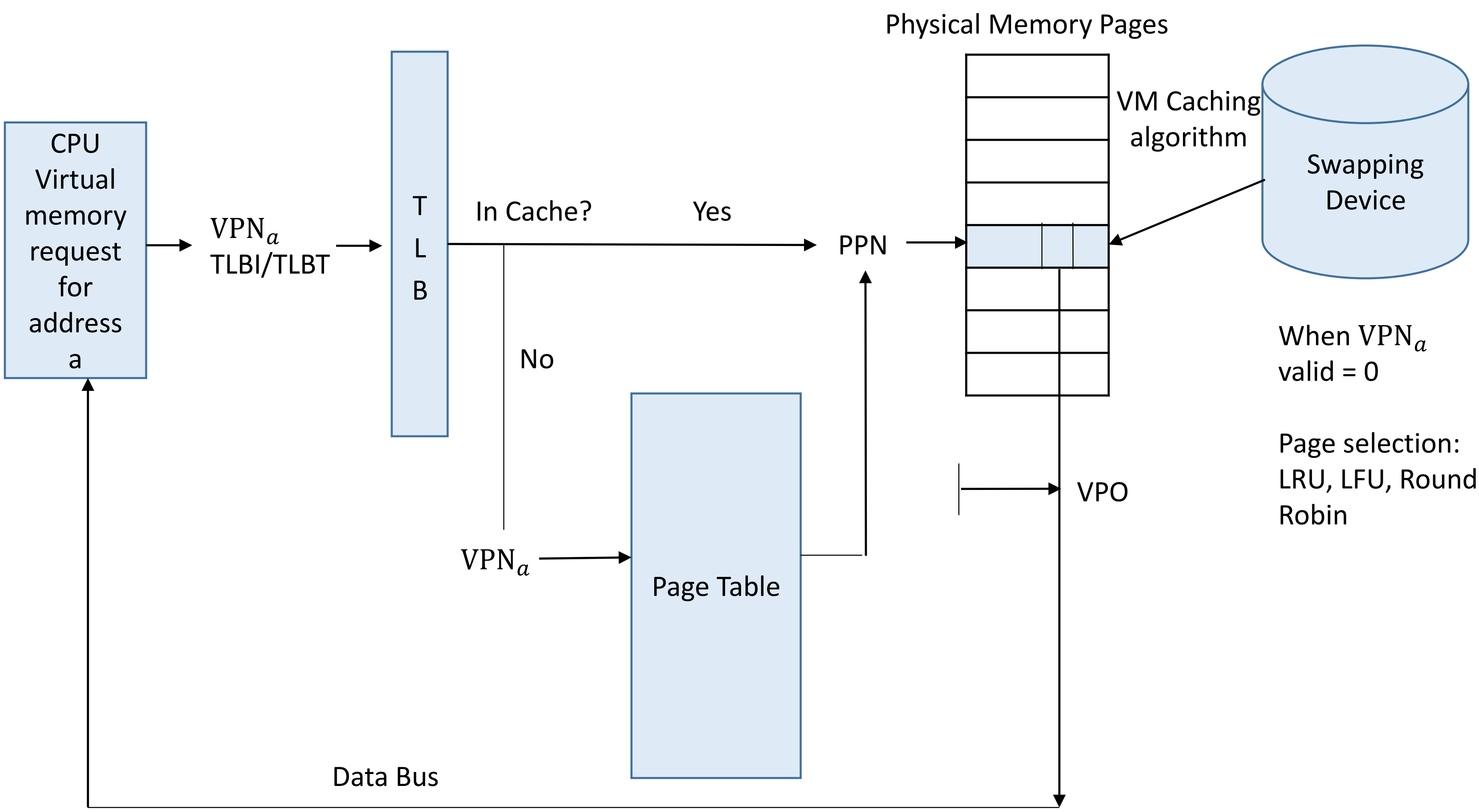
TLBI

Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
		0			0			0			0
		0	Tag	PTE	1			0			0
		0			0			0			0
		0			0			0			0
		0			0	Tag	PTE	1			0
		0			0			0			0
		0			0			0			0
Tag	PTE	1			0	TLBT	PTE	1			0
		0			0			0			0
		0			0			0			0
		0			0			0			0
		0			0			0			0
		0			0			0			0
		0			0			0			0
		0			0			0	Tag	PTE	1
		0			0			0			0
		0			0			0			0

Found by associativity with valid and tag in line based on TLBI

Virtual Memory

Address retrieval now looks like



When VPN_a valid = 0

Page selection: LRU, LFU, Round Robin

One Final Step

Once the physical address is obtained, need to retrieve from physical memory: L1 cache

Direct mapped. Say 8 blocks per line, 32 lines

Bits Physical Address

19			. . .								8	7				3	2		0
t	t	t	t	t	t	t	t	t	t	t	t	i	i	i	i	i	o	o	o

t's form a 10 bit unsigned number = Cache Tag

i's form a 5 bit unsigned number = Cache Index

o's form a 3 bit unsigned number = Cache Offset

L1 example

CI

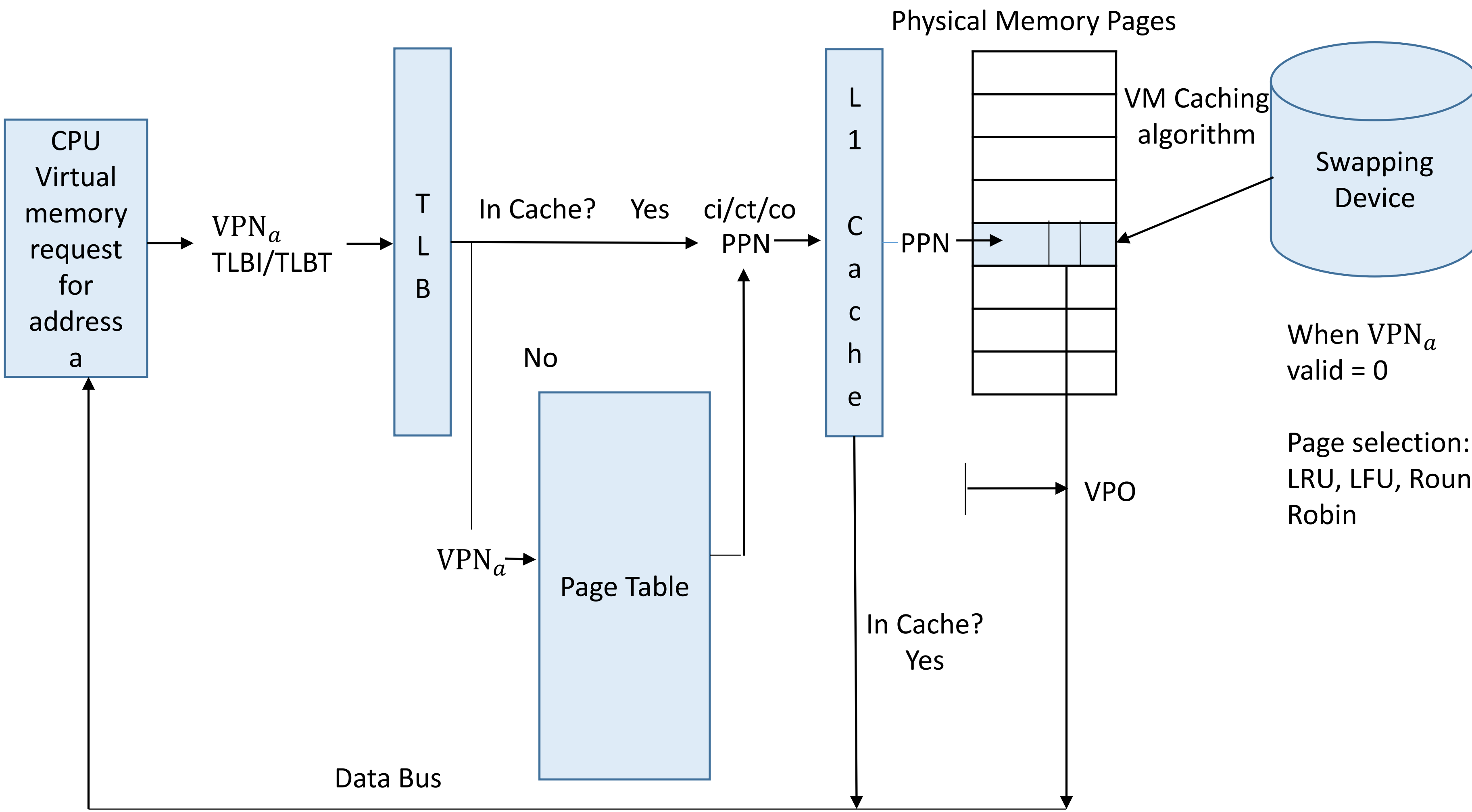
Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3	Blk 4	Blk 5	Blk 6	Blk 7
	0								
	0								
	0								
Tag	1	$Data_0$	$Data_1$	$Data_2$	$Data_3$	$Data_4$	$Data_5$	$Data_6$	$Data_7$
	0								
	0								
	0								
CT	1	$Data_0$	$Data_1$	$Data_2$	$Data_3$	$Data_4$	$Data_5$	$Data_6$	$Data_7$
	0								
	0								
	0								
Tag	1	$Data_0$	$Data_1$	$Data_2$	$Data_3$	$Data_4$	$Data_5$	$Data_6$	$Data_7$
	0								
	0								
Tag	1	$Data_0$	$Data_1$	$Data_2$	$Data_3$	$Data_4$	$Data_5$	$Data_6$	$Data_7$
	0								
	0								

CO

Virtual Memory

L1 Cache Example

Address retrieval now looks like

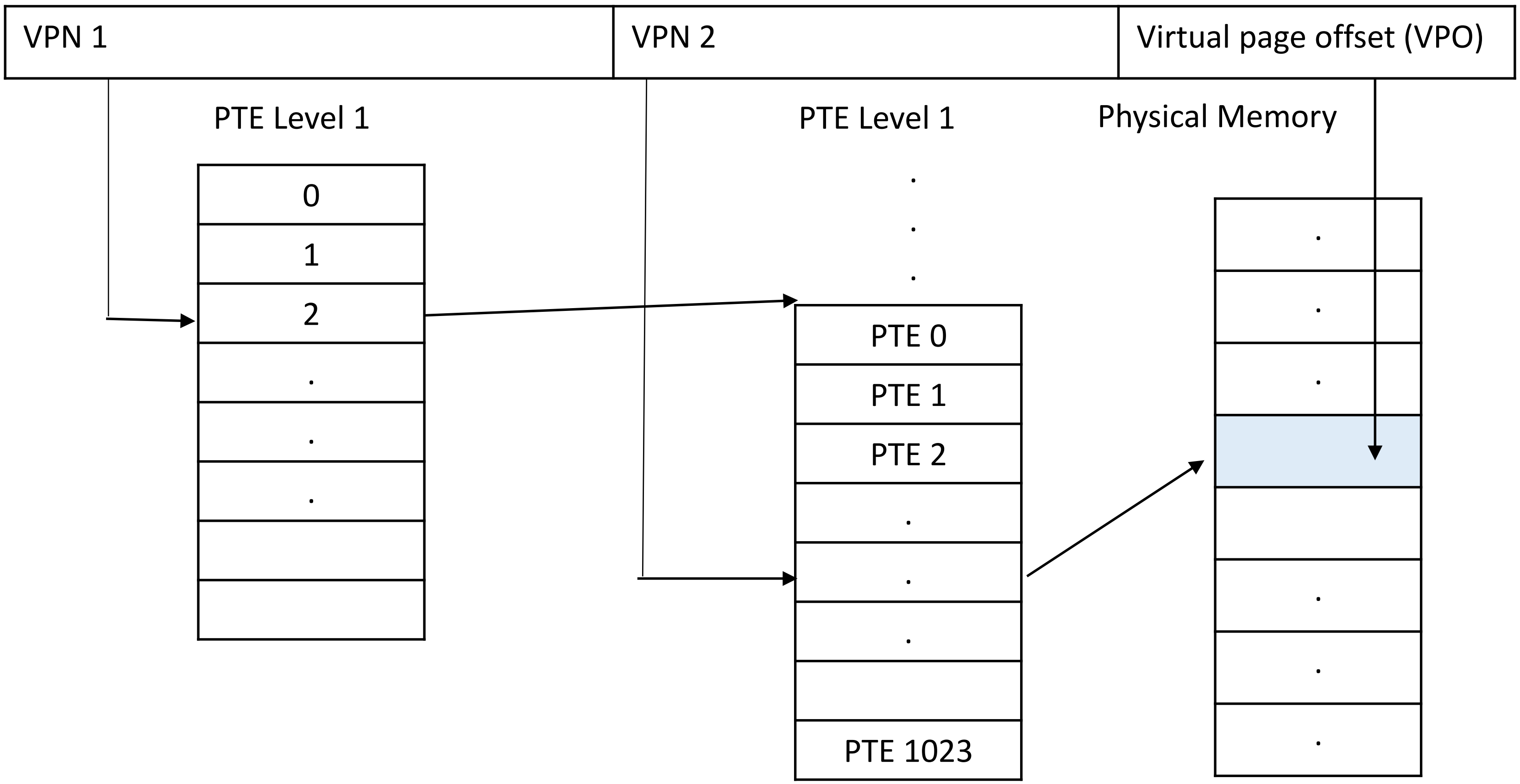


When VPN_a valid = 0

Page selection: LRU, LFU, Round Robin

Recall that we could have

corresponds to virtual address in its 4 MB section



Paths through MMU to retrieve address a.

Path 1: hit in TLB, PTE hit in cache

Step 1. Extract VPN and VPO of address

VPO = last p bits where 2^p = pages size
VPN = first n-p bits where there are 2^n virtual addresses

n= 14 p= 6 t= 2

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virt	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpo	vpo	vpo	vpo	vpo	vpo
TLB	tlbt	tlbt	tlbt	tlbt	tlbt	tlbt	tlbi	tlbi						

Step 2. Extract TLBI and TLBT from VPN

TLBI = last t bits where there are 2^t sets in TLB
TLBT = first n-t-p bits of VPN

Step 3. TLBT in TLB[TLBI] ? associatively ... valid ?

Paths through MMU to retrieve address a.

Step 4. retrieve PA = PPN concatenated with VPO if hit

Step 5. Extract CO from PA = last b bits where cache has 2^b blocks per set

Extract CI from PA = middle s bits where cache has 2^s sets

Extract CT from PA = first m-s-b bits where 2^m is physical memory size

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
cache			ct	ct	ct	ct	ct	ct	ci	ci	ci	ci	co	co
Phys			pn	pn	pn	pn	pn	pn	po	po	po	po	po	po

Step 6. cache[CI].valid and cache[CI].tag = CT ?

Step 7. return cache[CI].block[CO] as memory value

Virtual Memory

Paths through MMU to retrieve address a.

Path 2: hit in TLB, PTE miss in cache

Steps 1-5 for Path 1.

Step 6. ~cache[CI].valid or cache[CI] != CT

Step 7. Retrieve b blocks from memory into cache (direct mapped)

Step 8. return cache[CI].block[CO] as memory value

Path 3: miss in TLB

Steps 1-2 for Path 1.

Step 3. TLBT not in TLB[TLBI] ? associatively ... or ~valid ?

Step 4. Fetch PTE from memory

Step 5. Place appropriate PTEs in TLB

Steps 3-7 for path 1. ***Note*** cache miss still possible

Paths through MMU to retrieve address a.

Path 2: hit in TLB, PTE miss in cache

Steps 1-5 for Path 1.

Step 6. ~cache[CI].valid or cache[CI] != CT

Step 7. Retrieve b blocks from memory into cache (direct mapped)

Step 8. return cache[CI].block[CO] as memory value

Path 3: miss in TLB

Steps 1-2 for Path 1.

Step 3. TLBT not in TLB[TLBI] ? associatively ... or ~valid ?

Step 4. Fetch PTE from memory

Step 5. Place appropriate PTEs in TLB

Steps 3-7 for path 1. ***Note*** cache miss still possible

Example from text

n = 14 bit virtual address
m = 12 bit physical address
p = 6 = 64 byte pages
t = 2 = 4 sets in TLB, 4x associative (4 lines per set)
L1 cache 16 sets, 4 blocks

Looking for data at address 0x03d4

Worksheet 1

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virt	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpo	vpo	vpo	vpo	vpo	vpo
TLB	tlbt	tlbt	tlbt	tlbt	tlbt	tlbt	tlbi	tlbi						
0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	0	0

TLB Tag 0x03, TLB Index = 0x3, VPN = 0x0F, VPO = 0x14

Translation Lookahead Buffer

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

TBLI

TBLT

PPN = 0x0D = 1101_2, VPO = 0x14 = 010100_2

Concatenated = 1101010100 = 0x354

PPN to cache

Worksheet 2

Bit	11	10	9	8	7	6	5	4	3	2	1	0
Phys	ct	ct	ct	ct	ct	ct	ci	ci	ci	ci	co	co
0x0354	0	0	1	1	0	1	0	1	0	1	0	0

CT = 0x0d, CI = 0x05, CO = 0

Virtual Memory

L1 Cache

Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

CI

CT

Valid

CO

Data Value = 0x36

Virtual Memory

Example from text

n = 14 bit virtual address
m = 12 bit physical address
p = 6 = 64 byte pages
t = 2 = 4 sets in TLB, 4x associative (4 lines per set)
L1 cache 16 sets, 4 blocks

Looking for data at address 0x03d7

<u>Worksheet 1</u>

Bit	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Virt	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpn	vpo	vpo	vpo	vpo	vpo	vpo
TLB	tlbt	tlbt	tlbt	tlbt	tlbt	tlbt	tlbi	tlbi						
0x03d4	0	0	0	0	1	1	1	1	0	1	0	1	1	1

TLB Tag 0x03, TLB Index = 0x3, VPN = 0x0F, VPO = 0x17

Translation Lookahead Buffer

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
TBLI → 3	07	–	0	03	0D	1	0A	34	1	02	–	0

TBLT

PPN = 0x0D = 1101_2, VPO = 0x17 = 010111_2

Concatenated = 1101010111 = 0x357

Virtual Memory

PPN to cache

Worksheet 2

Bit	11	10	9	8	7	6	5	4	3	2	1	0
Phys	ct	ct	ct	ct	ct	ct	ci	ci	ci	ci	co	co
0x0354	0	0	1	1	0	1	0	1	0	1	1	1

CT = 0x0d, CI = 0x05, CO = 3

L1 Cache

Idx	Tag	Valid	Blk 0	Blk 1	Blk 2	Blk 3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

CI

CT

Valid

CO

Data Value = 0x1D

Dynamic Memory Allocation

Heap Management

Allocate memory when needed. Especially if amount not know at run time

2 types: explicit: must free unused space
implicit: used and unused space recognizable. free by garbage collection

Linux/C/C++ use explicit

void *malloc(int size in bytes) ; returns pointer to block: uninitialized. aligned on 16 byte

void *calloc(int size of word, int number of words) ; same as malloc except area is initialized.

void *realloc(void *ptr, tin new size in bytes) ; changes the size of an existing block

void *sbrk(int increment the heap size by) ;

void free(void *ptr) ; frees unused space. *ptr must equal a value obtained from malloc/calloc.

Dynamic Memory Allocation

Arbitrary sequences

of malloc, free

p1 = malloc(4) ;

p1	p1	p1	p1																

p2 = malloc(5) ;

p1	p1	p1	p1	p2	p2	p2	p2	p2											

p3 = malloc(8) ;

p1	p1	p1	p1	p2	p2	p2	p2	p2	p3	p3	p3	p3	p3	p3	p3	p3			

free(p3)

p1	p1	p1	p1						p3	p3	p3	p3	p3	p3	p3	p3			

p4 = malloc(2)

p1	p1	p1	p1	p4	p4				p3	p3	p3	p3	p3	p3	p3	p3			

Dynamic Memory Allocation

Characteristics

1. arbitrary sequences
2. immediate respone
3. use the heap
4. alignment
5. only manipulate free area

throughput: speed of the allocator

utilization: estimate of the amount of wasted space as a proportion of the total heap size.

largest total allocated space divided by the heap size.

Fragmentation. If the total free space is n and a request which is smaller than n cannot be satisfied, the heap is fragmented (external). If the request size needs to be rounded up, then there is wasted space (internal).

Final Exam Topics

Tuesday, December 16, 2014, 8-11AM LAKRETZ 110: open condensed notes, calculators OK.

Data representation

- Binary integers (base 10 to binary, etc)
- hexadecimal
- convert from one size to another (zero/sign extension)
- two's complement
- floating point
- integer, floating point addition multiplication, overflow
- boolean ops bitwise vs logical, precedence

Assembly lang

- operand spec.
- instruction coding: dst, src
- stack frame
- structs, arrays, unions

Optimization

- calls out of loops
- local vs call by name
- loop unroll
- blocking

Final Exam Topics

Memory

- Disks
- Spatial, temporal locality
- Cache
  - how works
  - direct mapped
  - associative (E > 1)
  - write issues

Exceptional control flow

- exceptions
- concurrent flows
- fork(), wait_pid, scheduling
- non local jumps

Final Exam Topics

Virtual Memory

- VM caching
  - hits, misses
  - page tables
  - page allocation
  - address translation

- dynamic memory allocation
  - allocator performance
  - allocation
    - placement (fit algorithms)
    - splitting
  - freeing
    - coalescing
  - alternative free lists

Final Exam Topics

Concurrent programming

- processes v threads
  - memory models
  - scheduling
    - races/deadlocks
- reaping/detaching
- synchronization
  - semaphores
- parallelization

- i/o multplexing
- process/thread scheduling

Implementation issues

- how to keep track of free blocks
- which free block to use
- how to split into free and used
- how to coalesce

Implicit free list

Beginning of each block contains a header with the block size and whether it is free or allocated. Then we can jump through the list using the block size.

S0 = size of block 0, F0 = 0 if free, 1 if allocated

S0	F0					S1	F1						S2	F2					
0						S0							S0+S1			. . .			

Use terminator block S = 0, allocated as end of list

implicit free list requires n = # free + # allocated steps to search the list

Dynamic Memory Allocation

Implementation issues

Fit strategies: first fit

large free blocks at the end
many small blocks at the begininig

next fit

search starting from last block split

best fit

exhaustive search of free list

To split or not. Give the whole block. Could decide based on how bad the fit is.

Do not split: gives poor utilization but better throughput.

Dynamic Memory Allocation

Implementation issues

When blocks are freed, the allocated/free bit is set to free. It might be that there is a free block next to it. Can merge into a larger free block.

Two options:

Immediate coalescing: check whenever a block is freed to see if coalescing is possible. Adds time to the free process.

Deferred coalescing: wait until a request cannot be satisfied.

Implementation issues

Coalescing cases when freeing the middle block:

Case 1: no adjacent free block

a	a	a	becomes	a	f	a

Case 2: previous block is adjacent

f	a	a	becomes	f	a

Case 3: prior block is adjacent

a	a	f	becomes	a	f

Case 4: combination of 2 & 3

f	a	f	becomes	f

Dynamic Memory Allocation

Heap simulation: implicit free list

Can speed up coalescing by duplicating header at end of block

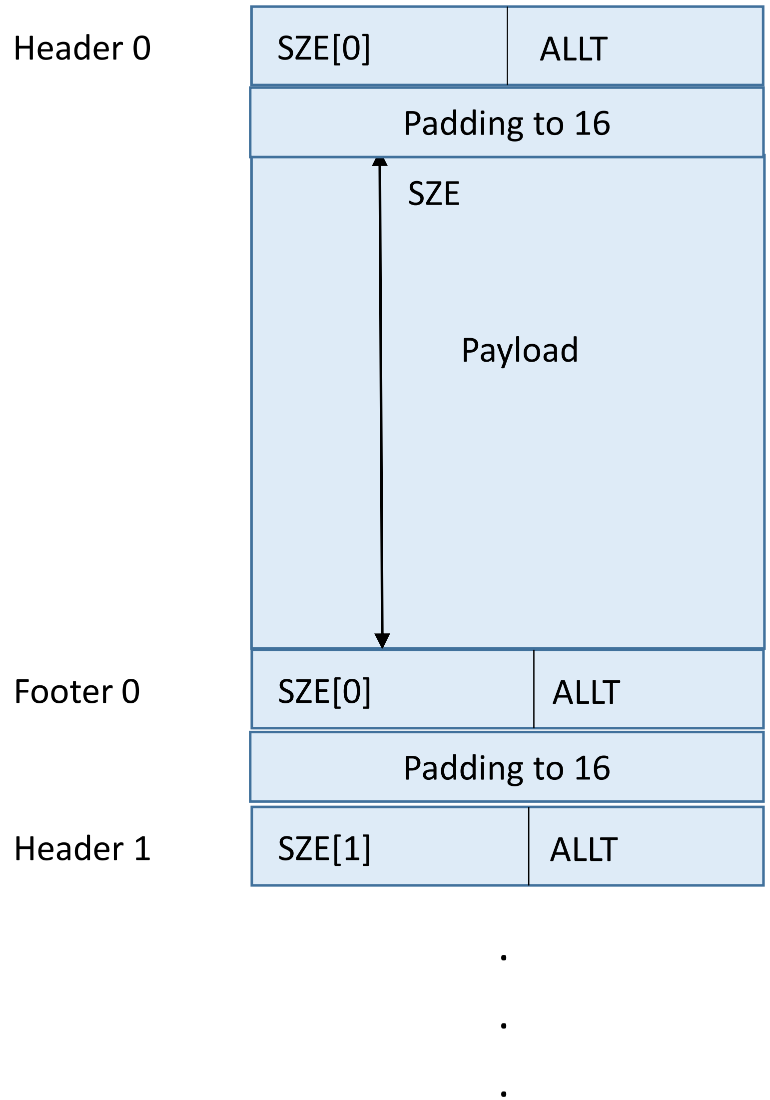
Let's make this more concrete by building up our own simulator:

halloc.c:

```
#define sze(i)  ((struct heap_hdr *) &heap[i]) -> SZE
#define allt(i) ((struct heap_hdr *) &heap[i]) -> ALLT

#define HEAPSIZE 32768

struct heap_hdr
  {
  short SZE  ;    // size of block (including headers)
  char  ALLT ;    // is this block allocated
  } ;

char *heap ;
```

Dynamic Memory Allocation

heap layout



SZE: size of block not including header/footer, padded to 16 byte.

ALLT(0 or 1): is this block free or allocated?

Header duplicated

heap simulation

Dynamic Memory Allocation

```
init_heap() ;
dump_heap( "init heap" ) ;

  for( i=0; i<14; i++ )
    blks[j++] = lalloc( rand()%500 ) ;
  dump_heap( "first allocs" ) ;

  lfree( &blks[ 9] ) ;
  lfree( &blks[10] ) ;
  dump_heap( "coalesce with lower (free 9 then 10)" ) ;

  lfree( &blks[ 6] ) ;
  lfree( &blks[ 5] ) ;
  dump_heap( "coalesce with upper (free 6 then 5)" ) ;

  lfree( &blks[1] ) ;
  lfree( &blks[3] ) ;
  lfree( &blks[2] ) ;
  dump_heap( "coalesce with both (free 1 then 3 then 2)" ) ;

  blks[5] = lalloc( 80000 ) ;
  dump_heap( "blow the top off (allocate 80,000)" ) ;

  for( i=0; i<14; i++ )
    if (blks[i] != 0 )
      {
      lfree( &blks[i] ) ;
      }
  dump_heap( "free everything " ) ;
```

```
init heap
  blk#    addr     sze     typ
     0       0   32736       0
         32752   32736       0
```

heap simulation

Dynamic Memory Allocation

first allocs

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	400	1
	832	400	1
2	848	288	1
	1152	288	1
3	1168	416	1
	1600	416	1
4	1616	304	1
	1936	304	1
5	1952	336	1
	2304	336	1
6	2320	400	1
	2736	400	1
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	432	1
	3920	432	1
10	3936	368	1
	4320	368	1
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

coalesce with lower (free 9 then 10)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	400	1
	832	400	1
2	848	288	1
	1152	288	1
3	1168	416	1
	1600	416	1
4	1616	304	1
	1936	304	1
5	1952	336	1
	2304	336	1
6	2320	400	1
	2736	400	1
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

Dynamic Memory Allocation

heap simulation

coalesce with lower (free 9 then 10)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	400	1
	832	400	1
2	848	288	1
	1152	288	1
3	1168	416	1
	1600	416	1
4	1616	304	1
	1936	304	1
5	1952	336	1
	2304	336	1
6	2320	400	1
	2736	400	1
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

coalesce with upper (free 6 then 5)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	400	1
	832	400	1
2	848	288	1
	1152	288	1
3	1168	416	1
	1600	416	1
4	1616	304	1
	1936	304	1
5	1952	768	0
	2736	768	0
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

Dynamic Memory Allocation

heap simulation

coalesce with upper (free 6 then 5)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	400	1
	832	400	1
2	848	288	1
	1152	288	1
3	1168	416	1
	1600	416	1
4	1616	304	1
	1936	304	1
5	1952	768	0
	2736	768	0
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

coalesce with both (free 1 then 3 then 2)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	1168	0
	1600	1168	0
4	1616	304	1
	1936	304	1
5	1952	768	0
	2736	768	0
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

Dynamic Memory Allocation

coalesce with both (free 1 then 3 then 2)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	1168	0
	1600	1168	0
4	1616	304	1
	1936	304	1
5	1952	768	0
	2736	768	0
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	28016	0
	32752	28016	0

blow the top off (allocate 80,000)

blk#	addr	sze	typ
0	0	384	1
	400	384	1
1	416	1168	0
	1600	1168	0
4	1616	304	1
	1936	304	1
5	1952	768	0
	2736	768	0
7	2752	496	1
	3264	496	1
8	3280	160	1
	3456	160	1
9	3472	832	0
	4320	832	0
11	4336	32	1
	4384	32	1
12	4400	192	1
	4608	192	1
13	4624	64	1
	4704	64	1
14	4720	80000	1
	84736	80000	1
15	84752	13520	0
	98288	13520	0

free everything

blk#	addr	sze	typ
0	0	98272	0
	98288	98272	0

Dynamic Memory Allocation

Explicit Free lists

Maintain a linked list of free areas: when looking for a block, do not need to go through the allocated blocks.

```
struct HDR      // free block header/footer/linked list
  {
  int  payload ;    // size of block (excluding headers)
  char freeall ;    // is this block allocated? 0=free/1=allocated
  int  succesr ;    // successor free block
  int  previus ;    // previous free block
  } anchor ;
```

anchor is the head of the free list. The header of each free block contains the same structure.

Dynamic Memory Allocation

Review Lab 4

Segregated free lists

More than one free list: segregated by size range: many variations.

Simple segregated free lists: all blocks in size range the same size:

always allocate entire block (no splitting, coalescing)
request new block, divide it into same size blocks

fast allocation, freeing
fragmentation danger

Segregated fits: lists segregated by size range

search list by size range: first fit. split block and put remnant on appropriate list
free: coalesce and place on size range list

close to best fit search

Buddy system: lists always point to powers of 2 size blocks. always round requests up to power of 2.

search list. if none available take larger block and split in two, repeatedly.
when size reached, a “buddy” is left over.
when free occurs, can coalesce with buddy.

fragmentation

List of common memory bugs

Passing value instead of pointer:

```
void f1( int *x ) {

  x = 20 ;
  }

void main() {
  int z ;

  f1( z ) ;  // instead if f1( &z ) ;
  }
```

Unitialized values:

```
void main() {
  int i, *a, b[5], c[5] ;

  int *a = (int *)malloc( 5*sizeof(int) ) ;
  for( i=0; i<5; i++ )
    c[i] = a[i]+b[i] ;

  }
```

List of common memory bugs

<ins>gets</ins>:

```
void f1( int *x ) {
  char b[10] ;

  gets( b ) ;
  }
```

<ins>sizeof(int) and sizeof(int *) not the same</ins>:

```
void f1( int n ) ;

  int **a = (int **) malloc( n*sizeof(int) ) ;  // instead of sizeof( int * )
 }
```

<ins>pointer instead of value referencing</ins>:

```
void f1( int *x ) {

*x-- ;  // instead of (*x)-- ;
}
```

Dynamic Memory Allocation

List of common memory bugs

<u>pointer arithmetic</u>:

```
void f1( int *x ) {

x += sizeof(int) // instead of x++ ;
}
```

<u>memory pointed to disappears</u>:

```
int *f1( int n ) ;
  int a ;

 return &a ;
 }
```

<u>access freed area</u>:

```
void f1( int *x ) {
int *y ;

y = (int *)malloc( 20*sizeof(int) ) ;
free y ;

for( i=0;i<20; i++ )
   x[i] = y[i] ;
}
```

Dynamic Memory Allocation

List of common memory bugs

memory leaks:

```
void f1() {
int *x ;

x = (int *) malloc( 100 ) ;
return ;
}
```

Items from chapter 11

int open_listenfd(int port) ;

Returns a “listening descriptor”, connecting the caller to “port”. -1 if error

int accept(int listenfd, struct sockaddr *addr, int *addrlen) ;

Waits for a connection on listenfd (the port opened).

example with processes

```
#include "csapp.h"
void echo(int connfd);

void sigchld_handler(int sig)  {
  while (waitpid(-1, 0, WNOHANG) > 0)  ;
  return;
  }

int main(int argc, char **argv)  {
   int listenfd, connfd, port;
   socklen_t clientlen=sizeof(struct sockaddr_in);
   struct sockaddr_in clientaddr;

  if (argc != 2) {
    fprintf(stderr, "usage: %s <port>\n", argv[0]);
    exit(0);  }
  port = atoi(argv[1]);

  signal(SIGCHLD, sigchld_handler);
  listenfd = open_listenfd(port);
  while (1) {
    connfd = Accept(listenfd, (SA *) &clientaddr, &clientlen);
    if (fork() == 0) {
      close(listenfd); /* Child closes its listening socket */
      echo(connfd); /* Child services client */
      close(connfd); /* Child closes connection with client */
      exit(0); /* Child exits */
      }
   Close(connfd); /* Parent closes connected socket (important!) */
   }  }
```

processes, descriptor sets

using fork() gives separate address space. creates safe environment but inhibits communication.

file descriptor set (remember wait set). An integer: fd_set, on bit for each file (port)

```
FD_ZERO( fd_set *fdset ) ;                  // zeros descriptor set
FD_CLR( int  fd, fd_set *fdset ) ;          // removes fd from descriptor set
FD_SET( int  fd, fd_set *fdset ) ;          // enters fd into descriptor set
FD_ISSET(int  fd, fd_set *fdset ) ;         // test fd on in descriptor set
```

```
int select( int n, fd_set *fdset, NULL, NULL, NULL ) ;
```

fdset is the descriptor set and n is the number of descriptors. BLOCKS until a file descriptor is ready. Returns the number of ready descriptors.

descriptor set example

```
                .
                .
                .

FD_ZERO(&read_set);                          /* Clear read set */
FD_SET(STDIN_FILENO, &read_set);             /* Add stdin to read set */
FD_SET(listenfd, &read_set);                 /* Add listenfd to read set */

while (1) {
  ready_set = read_set;
  select(listenfd+1, &ready_set, NULL, NULL, NULL);
  if (FD_ISSET(STDIN_FILENO, &ready_set))
    command();                               /* Read command line from stdin */
  if (FD_ISSET(listenfd, &ready_set)) {
    connfd = accept(listenfd, (SA *)&clientaddr, &clientlen);
    echo(connfd);                            /* Echo client input until EOF */
    close(connfd);
    }
  }
  }

void command(void) {
  char buf[MAXLINE];
  if (!Fgets(buf, MAXLINE, stdin))
    exit(0); /* EOF */
  printf("%s", buf);                         /* Process the input command */
  }
```

Threads

Different from processes. No children but “peers” sharing virtual memory.

Thread context: stack, instruction counter, registers and condition code register.

Rest is shared. Same scheduling procedure as processes. But context switch excludes VM.

Main thread: peer threads. less hierarchy.

Creation is by naming a function: typedef void *(func)(void *) ; // pointer to a function

int pthread_create(pthread_t *tid, pthread_attr_t *attr, func *f, void *arg) ;

Creates a thread using function f, sets thread ID into tid, returns 0 if OK, -1 if error. Ignore attr, for now.

pthread_t pthread_self() ;

returns the TID.

int pthread_join(pthread_t tid, void **thread_return) ;

Waits for tid to terminate and reaps the thread. can only wait for specific one.

Example

```
#include <pthread.h>
#include <stdio.h>
void *thread(void *vargp);

int main()
  {
  pthread_t tid;
  pthread_create(&tid, NULL, thread, NULL);
  pthread_join(tid, NULL);
  exit(0);
  }

void *thread(void *vargp) /* Thread routine */
  {
  printf("Hello, world!\n");
  return NULL;
  }
```

Thread control

void pthread_exit(void *thread_return) ;

waits for all peer threads (created by the caller) to terminate.

void pthread_cancel(pthread_t tid) ;

To end a thread:

implicit exit by returning
explicit: call pthread_exit, waiting for all peer threads to terminate
exit(): ends the entire process
call pthread_cancel for this TID (by another peer)

void pthread_detach(pthread tid) ;

“unpeers” the thread. Does not need top be reaped. Can no longer be cancelled. Can detach yourself.

pthread_once_t once_control = PTHREAD_ONCE_INIT ;
int pthread_once(pthread_once_t *once_control, void (*init routine)(void)) ;

Call the init routine the first time and skips all subsequent times.

Thread example vs process example

```
void echo(int connfd);
void *thread(void *vargp);

int main(int argc, char **argv)  {
  int listenfd, *connfdp, port;
  socklen_t clientlen=sizeof(struct sockaddr_in);
  struct sockaddr_in clientaddr;
  pthread_t tid;

  port = atoi(argv[1]);

  listenfd = Open_listenfd(port);
  while (1) {
    connfdp = malloc(sizeof(int));        // create an instance to prevent races
   *connfdp = accept(listenfd, (SA *) &clientaddr, &clientlen);
    pthread_create(&tid, NULL, thread, connfdp);
    } }
/* Thread routine */
void *thread(void *vargp) {
  int connfd = *((int *)vargp);
  pthread_detach(pthread_self());
  free(vargp);
  echo(connfd);
  close(connfd);
  return NULL;
  }
```

Shared variable issues in threads

```
#include <pthread.h>
#include <stdio.h>
#define N 2
void *thread(void *vargp);
char **ptr; /* Global variable */

int main()
  {
  long int i;
  pthread_t tid, pret;
  char *msgs[N] = {
      "Hello from foo",
      "Hello from bar"
        };

  ptr = msgs;
  for (i = 0; i < N; i++)
    pthread_create(&tid, NULL, thread, (void *)i);

  pthread_exit(NULL);
  }

void *thread(void *vargp)
  {
  long int myid = (long int)vargp;
  static int cnt = 0;
  printf("[%d]: %s (cnt=%d)\n", myid, ptr[myid], ++cnt);
  return NULL;
  }
```

C storage classes:

Global variables – one instance
Local automatic variables (stack variables)
Local static variables

Shared variables have only one instance

ptr is a global variable
msgs is local automatic but shared by aliasing
myid is local automatic
cnt is local static

Counting threads

```
#include <pthread.h>
void *thread(void *vargp); /* Thread routine prototype */

volatile int cnt = 0; /* Counter */

int main(int argc, char **argv)
  {
  int niters = 1000000 ;
  pthread_t tid1, tid2;

  /* Create threads and wait for them to finish */
  pthread_create(&tid1, NULL, thread, &niters);
  pthread_create(&tid2, NULL, thread, &niters);
  pthread_join(tid1, NULL);
  pthread_join(tid2, NULL);

  /* Thread routine */
  void *thread(void *vargp)
  {
  int i, niters = *((int *)vargp);

  for (i = 0; i < niters; i++)
    cnt++;
  }
```

Shared variable issues in threads

show threads1.c

```
                       the loop:                                                       Thread 1    Thread 2
40069f <+11>:  mov   0x2005ab(%rip),%edx     # 0x600c50 <cnt>   //        1           4
4006a5 <+17>:  add   $0x1,%edx                                  //        2           5
4006a8 <+20>:  mov   %edx,0x2005a2(%rip)     # 0x600c50 <cnt>   //        3           6
4006ae <+26>:  add   $0x1,%eax
4006b1 <+29>:  cmp   %ecx,%eax
4006b3 <+31>:  jne   0x40069f <thread+11>
```

For thread 1, we have operations 1, 2 and 3, thread 2 , operations 4, 5 and 6

123123	142312	412312	451231
123124	142315	412315	451236 *
123142	142351	412351	451263 *
123145	142356 *	412356 *	451264
123412	142531	412531	451623 *
123415	142536 *	412536 *	451624
123451	142563 *	412563 *	451642
123456 *	142564	412564	451645
124312	145231	415231	456123 *
124315	145236 *	415236 *	456124
124351	145263 *	415263 *	456142
124356 *	145264	415264	456145
124531	145623 *	415623 *	456412
124536 *	145624	415624	456415
124563 *	145642	415642	456451
124564	145645	415645	456456

Label %edx with thread #,
%edx1 = thread 1's %edx
%edx2 = thread 2's %edx

cnt = 0

123456 means:

```
mov   cnt,%edx1
add   $0x1,%edx1
mov   %edx1,cnt
mov   cnt,%edx2
add   $0x1,%edx2
mov   %edx2,cnt
```

cnt = 2

cnt = 0

124356 means:

```
mov   cnt,%edx1
add   $0x1,%edx1
mov   cnt,%edx2
mov   %edx1,cnt
add   $0x1,%edx2
mov   %edx2,cnt
```

cnt = 1

Semaphores

Need to “reserve” shared variables to protect.

```
int sem_init( sem_t *x, 0, unsigned int value) ;    // initializes to value (realy just an int)

int sem_wait( sem_t x ) ;                           // reserve resource (decrement x)
                                                    // if 0, wait until ready

int sem_post( sem_t x ) ;                           // release resource (increment x)
                                                    // if > 0, start first waiting thread
```

While x is shared, incrementing, decrementing, testing done in uniterruptible mode.

Semaphore an integer.

show threads4.c

Counting Semaphores

can also initialize a semaphore, n, to something > 1.

Then n is a counter. waiting decrements (if not zero, continue, if zero, wait), posting increments. Also, posting a semaphore multiple times

Producer-Consumer: have a display case with product. if not full, a producer can place a product on the shelf, if full, must wait for a space. a consumer can take a product off the shelf, if empty must wait for a product. Need to synchronize because the shelf is a shared object.

show semaphore1.c

Readers:Writers: When reading an item, no changes are made so it is OK to have multiple readers. But when writing, must take exclusive control. Readers over writers: if anyone is reading, the writer must wait, even if more readers come along. Writers over readers, once a writer comes, no new readers can come until the writer is finished.

show semaphore2.c

Parallelism with threads

Concurrency vs. parallelism revisited

one core vs multiple cores

Create threads to take advantage of cores. But must look out for shared variables.

show cores.c

What is the difference w.r.t process vs threads for multiple cores: context change.

show coref.c

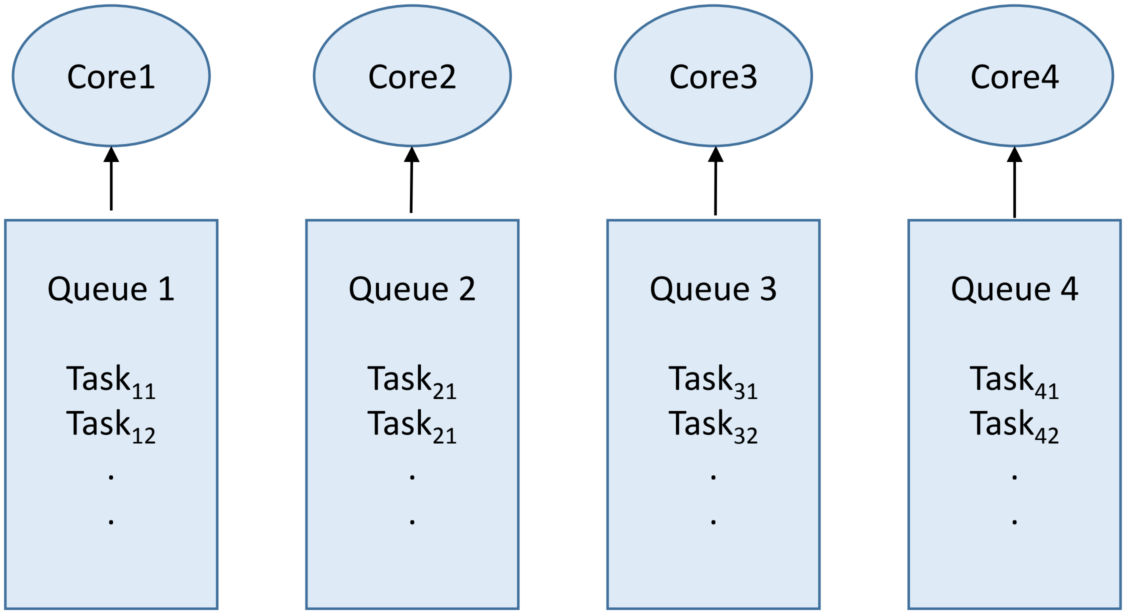
What about competing threads?

show corew.c

show cores.qpw

Scheduling?

Single core: one queue. What about renegade processes: while(1) {} ?? Time slicing



Indifferent whether process or thread. Context switch has more overhead for processes

CPU affinity

Other Issues

"Thread Safety": guarantees that afunction always gives the correct answer in a concurrent, threaded environment.

Class 1: does not protect read/write shard variables
2: keeps state in non-automatic variables
3: return pointers to static variables
4: functions which call unsafe functions

Re-entrant functions: shared library routines cannot reference shared read/write data

Races: having the correct value of a shared value depends on who gets there first.

Deadlocks (deadly embrace):

Thread A holds resource x, B holds y, A needs y but B needs x inorder to relase y.

Thread A:	Thread B:
pthread_wait(&x) ;	pthread_wait(&y) ;
pthread_wait(&y);	pthread_wait(&x);
	ptrhead_post(&y) ;

Rule of thumb: all threads must wait and post multiple resources in the same order.
Here: A waits x then y, B waits y then x.

Concurrent Programming

Threads, cores and semaphores recap

Counting thread adds 1 to a global variable:

```
/* Thread routine */
void *thread(void *vargp)
{
int i, niters = *((int *)vargp);

for (i = 0; i < niters; i++)
cnt++;                          //  cnt a global variable

return NULL;
}
```

Run it with random size niter, increasing number of threads

metric: cnt/(niter x number of threads) is >= 1/(number of threads), <= 1.000

metric = 1/(number of threads) means that cnt = niter (addition wrong every time)

metric = 1 means that cnt = niter x (number of threads) (addition correct every time)

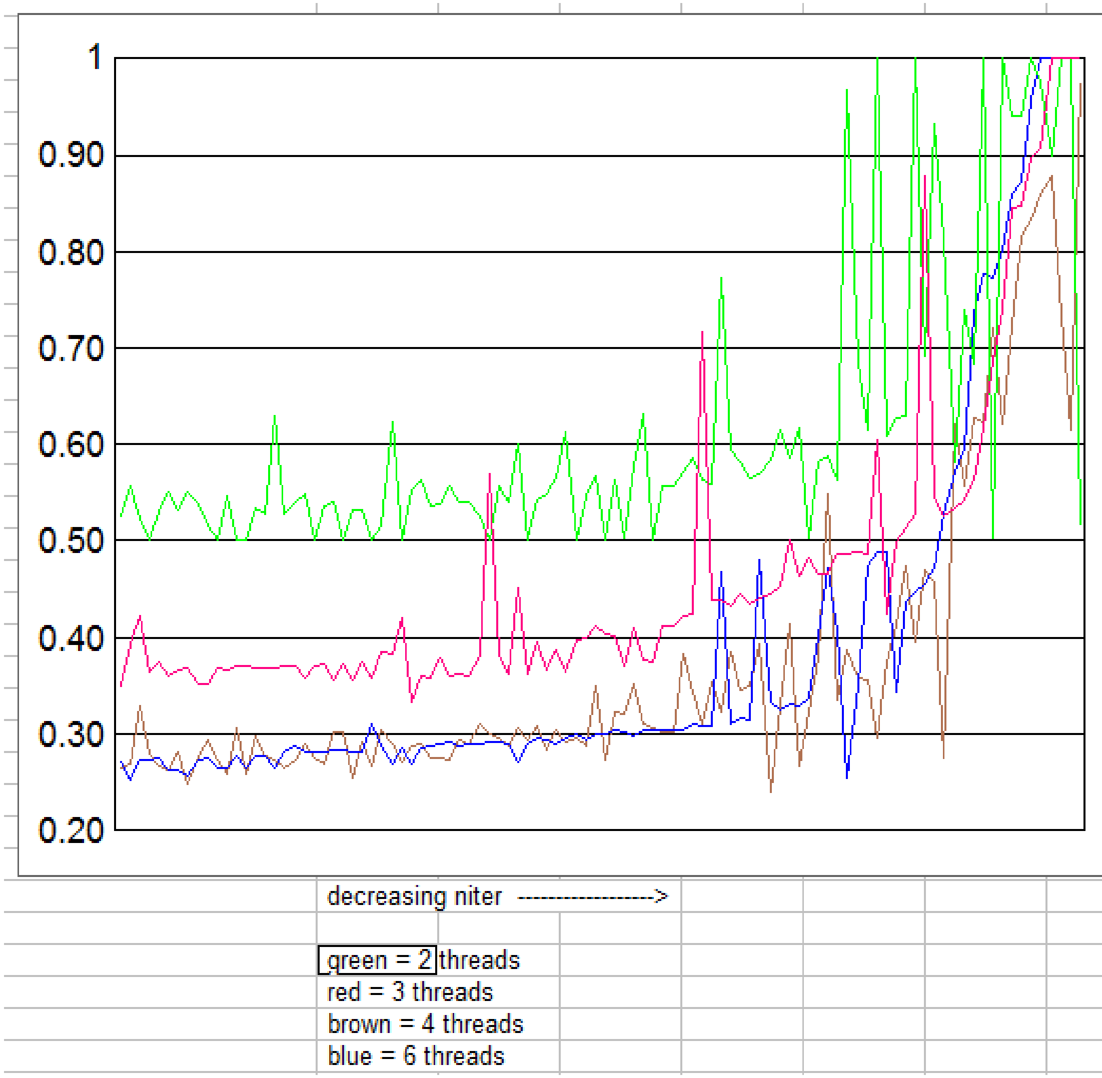
Threads conclusions

Conclusions:

1) For large niter, the metric is 1/(number of threads), up to 5.

	Metric
1000000 2 x niter=2000000 cnt=1006491	0.503
1000000 2 x niter=2000000 cnt=1000073	0.500
1000000 2 x niter=2000000 cnt=1003433	0.502
1000000 3 x niter=3000000 cnt=1021719	0.341
1000000 3 x niter=3000000 cnt=1001199	0.334
1000000 3 x niter=3000000 cnt=1006640	0.336
1000000 4 x niter=4000000 cnt=1013463	0.253
1000000 4 x niter=4000000 cnt=1013291	0.253
1000000 4 x niter=4000000 cnt=1092506	0.273
1000000 5 x niter=5000000 cnt=1057547	0.212
1000000 5 x niter=5000000 cnt=1000114	0.200
1000000 5 x niter=5000000 cnt=1022921	0.205
1000000 6 x niter=6000000 cnt=1730981	0.288
1000000 6 x niter=6000000 cnt=1776257	0.296
1000000 6 x niter=6000000 cnt=1588655	0.265

2) For decreasing niter, metric goes to 1.



Threads conclusions

Renegade thread:

```
/* Thread routine */
  void *while1(void *vargp)
  {
  while( 1 )
    {
    }

  return NULL;
  }
```

Starting renegade threads in addition to the counting threads has little effect

```
1000000 4 x niter=4000000 cnt=1524595   0.381
1000000 4 x niter=4000000 cnt=1298805   0.325
1000000 4 x niter=4000000 cnt=1279227   0.320
1000000 4 x niter=4000000 cnt=1266988   0.317
1000000 5 x niter=5000000 cnt=1073560   0.215
1000000 5 x niter=5000000 cnt=1364531   0.273
1000000 5 x niter=5000000 cnt=1142935   0.229
1000000 5 x niter=5000000 cnt=1392169   0.278
1000000 5 x niter=5000000 cnt=1296054   0.259
1000000 6 x niter=6000000 cnt=1724155   0.287
1000000 6 x niter=6000000 cnt=1444135   0.241
1000000 6 x niter=6000000 cnt=1311778   0.219
1000000 6 x niter=6000000 cnt=1241825   0.207
1000000 6 x niter=6000000 cnt=1297965   0.216
```

Threads/parallelism recap

Parallel counting: create threads to add up 1-n (n very large) by dividing into sectors and have each thread add up one sector:

```
void *sum(void *vargp)
  {
  int i = *((int *)vargp) ;
  long long int j,k,l,a = 0 ;

  k = (i-1)*sector_size   ;
  l =  i   *sector_size-1 ;
  for( j=k; j<l; j++ )
    a = a+j ;

  tot[i] = a ;
  }
```

If T = the time to add up 1-n, then T/n should be the time to add the numbers with n threads: up to the number of cores. But we get some surprising results

Threads	Sector Size	T	T/T_0	1/Threads
1	1073741824	0.807	1.000	1.000
2	536870912	0.414	0.513	0.500
3	357913941	0.272	0.337	0.333
4	268435456	0.205	0.254	0.250
5	214748364	0.171	0.212	0.200
6	178956970	0.143	0.177	0.167
7	153391689	0.125	0.155	0.143
8	134217728	0.107	0.132	0.125
9	119304647	0.152	0.188	0.111

Improvement continues after 4 (number of cores on server)

Threads vs Processes

Can do the same test using processes instead of threads. get close to the same results.

Creating large working set in the process slows things down a little.

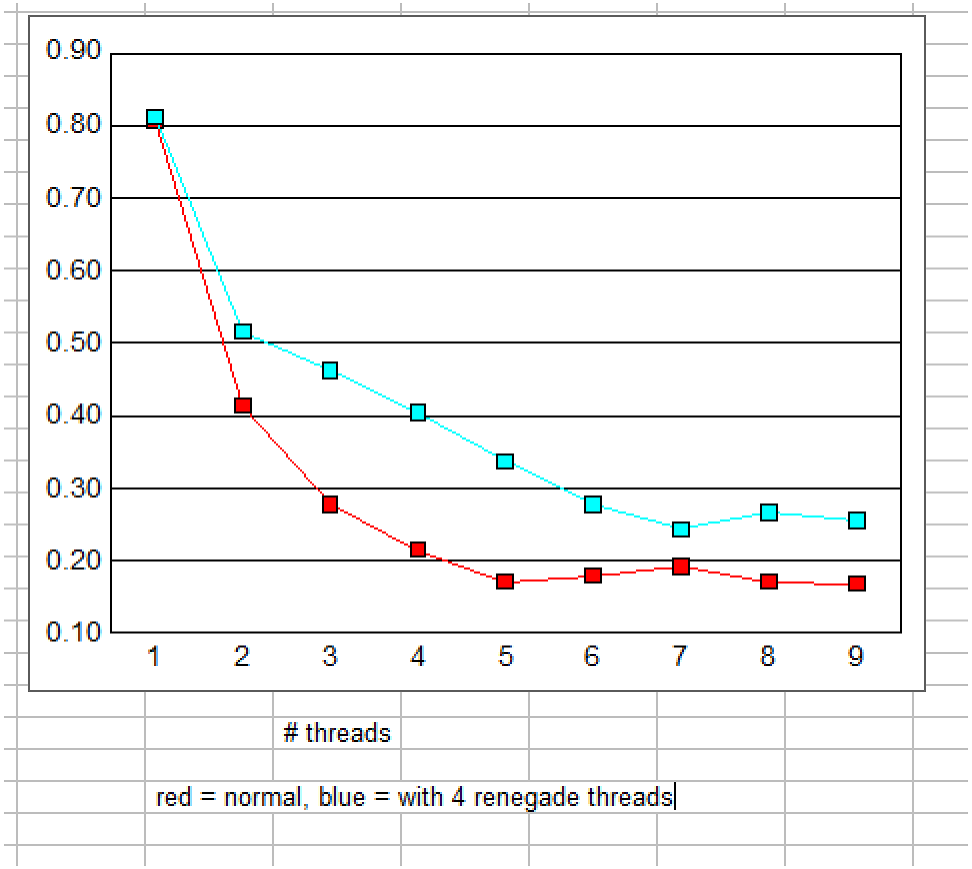
Threads	Sector Size	T	T/T_0	1/Threads
1	1073741824	0.817	1.000	1.000
2	536870912	0.421	0.516	0.500
3	357913941	0.288	0.353	0.333
4	268435456	0.222	0.272	0.250
5	214748364	0.179	0.219	0.200
6	178956970	0.189	0.231	0.167
7	153391689	0.197	0.241	0.143
8	134217728	0.182	0.223	0.125
9	119304647	0.179	0.220	0.111

Adding 4 renegade threads has an impact but reasons are not explainable.

Threads	Sector Size	T	T/T_0	1/Threads
1	1073741824	0.850	1.000	1.000
2	536870912	0.426	0.500	0.500
3	357913941	0.481	0.566	0.333
4	268435456	0.383	0.451	0.250
5	214748364	0.313	0.368	0.200
6	178956970	0.282	0.332	0.167
7	153391689	0.243	0.286	0.143
8	134217728	0.237	0.279	0.125
9	119304647	0.230	0.270	0.111

Renegade threads

Compare with renegade threads



Semaphore programs

Producer/consumer simulation

```
sem_init( &x, 0, 1 ) ;  //  display case lock
sem_init( &n, 0, 5 ) ;  //  empty spaces
sem_init( &d, 0, 0 ) ;  //  filled spaces

void *produce(void *vargp)    {
  sem_wait( &n ) ;   // waits if full ( n = 0 )
  sem_wait( &x ) ;   //  locks case
  printf( "Produce %2d %2d %2d\n", i, n.__align, d.__align+1 ) ;
  sem_post( &x ) ;   // unlocks case
  sem_post( &d ) ;   // produces
  return NULL;
  }

void *consume(void *vargp)    {
  sem_wait( &d ) ;   // waits if empty ( d = 0 )
  sem_wait( &x ) ;   //  locks case
  printf( "Consume %2d %2d %2d\n", i, n.__align+1, d.__align ) ;
  sem_post( &x ) ;   // unlocks case
  sem_post( &n ) ;   // consumes
  return NULL;
  }
```

CCPPPCCCPPPCPCPCCPP

	s	e	f
Produce	1	4	1
Consume	1	5	0
Produce	2	4	1
Produce	3	3	2
Produce	4	2	2
Produce	5	1	2
Consume	5	2	0
Consume	3	3	0
Consume	4	4	0
Consume	2	5	0
Produce	6	4	1
Consume	6	5	0
Produce	7	4	1
Consume	7	5	0
Produce	8	4	1
Consume	8	5	0
Produce	9	4	1
Consume	9	5	0
Produce	10	4	1

s = sequence
e = empty spaces
f = full spaces

Semaphore programs

Reader/writer simulation

```
  sem_init( &x, 0, 1 ) ; // update readers count lock
  sem_init( &n, 0, 1 ) ; // writer lock

void *reader(void *vargp)   {
  sem_wait( &x ) ;  // lock readers
  readers++ ;
  if( readers == 1 )   // wait on writer if first reader
    sem_wait( &n ) ;
  sem_post( &x ) ;

  sem_wait( &x ) ;  lock readers count
  readers-- ;
  if( readers == 0 )   / if last reader, let writer go
    sem_post( &n ) ;
  sem_post( &x ) ;
  return NULL;
  }
void *writer(void *vargp) {
  {
  sem_wait( &n ) ; // wait for readers
  sem_post( &n ) ; // let readers go
  return NULL;
  }
```

WRRWWWRRRWRWRWRRRRW

	s	r	w
Writer start	1	0	1
Reader start	1	1	0
Reader end	1	0	1
Writer start	2	0	1
Reader start	2	1	0
Reader end	2	0	1
Writer start	3	0	1
Reader start	3	1	0
Reader end	3	0	1
Writer start	4	0	1
Reader start	4	1	0
Reader end	4	0	1
Reader start	5	1	0
Reader end	5	0	1
Writer start	5	0	1
Reader start	6	1	0
Reader start	7	2	0
Reader end	6	1	0
Reader end	7	0	1
Reader start	8	1	0

s = sequence
r = # readers
w = writer locked

Reader/writer observations

Process highly influenced by amount of time reader takes.
Introduce sleep(1) in reader, delays all writers.

Assembly Language Review

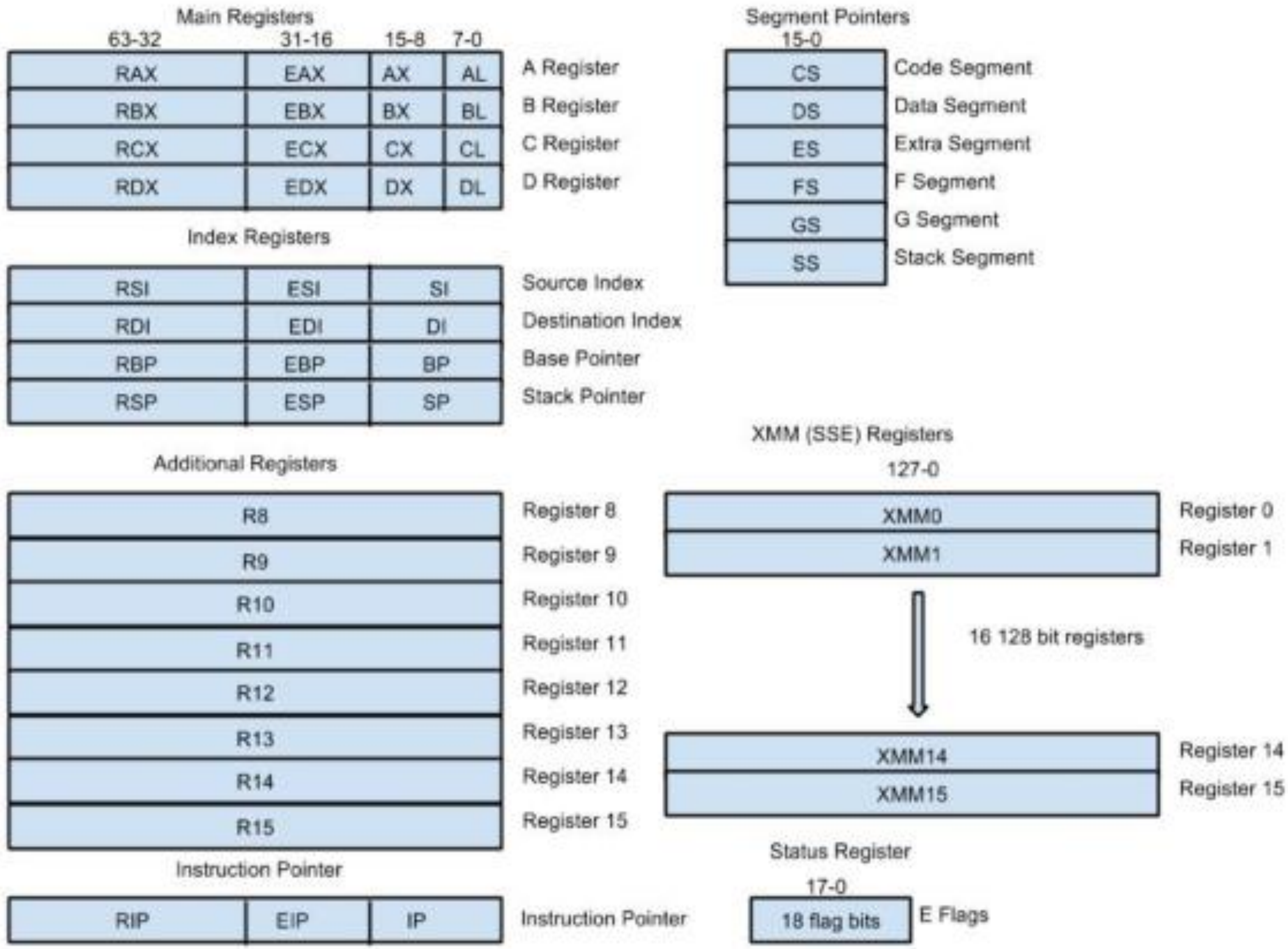
C Data Types

Type	bytes (X68-64)	mnemonic	description
char	1	b	Character (can also be interpreted as integer)
short	2	w	Short integer
int	4	1	Integer
long int	8	q	Long integer
long long int	8	q	Long integer
char *	8	q	Pointer
float	4	s	Single precision floating point
double	8	d	Double precision floating point
long double	16	t	Extended precision floating point

No bit data type

b	byte (8 bits)
w	word (16 bits)
l	long (32 bits)
q	quad (64 bits)
s	single precision (32 bits)
d	double precsions (64 bits)
t	extended precision (128 bits)

X86-64 registers



How to specify an operand

% implies a register — %rax, %eax, %ax, %al
(64) (32) (16) (8)

$ means "immediate" or exactly that value — $0x5: the value 5: 0x means hexadecimal

parentheses means the value stored at that memory address — (%rax)

Type	Form	Operand value	Name
Immediate	$num	num	Immediate
Register	%rax	%rax	Register
Memory	num	(num)	Absolute
Memory	(%rax)	(%rax)	Indirect
Memory	num(%rax)	(num+%rax)	Base+displacement
Memory	(%rax,%rbx)	(%rax+%rbx)	Indexed
Memory	num(%rax,%rbx)	(num+%rax+%rbx)	Indexed
Memory	(,%rax,s)	(%rax*s)	Scaled indexed
Memory	num(,%rax,s)	(num+%rax*s)	Scaled indexed
Memory	(%rax,%rbx,s)	(%rax+%rbx*s)	Scaled indexed
Memory	num(%rax,%rbx,s)	(num+%rax+%rbx*s)	Scaled indexed

Note: s may only be 1, 2, 4 or 8

Operand examples

Assembly Language Review

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	ff	fe	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0
1	fe	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef
2	fd	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee
3	fc	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed
4	fb	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec
5	fa	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb
6	f9	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea
7	f8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9
8	f7	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9	e8
9	f6	f5	f4	f3	f2	f1	f0	ef	ee	ed	ec	eb	ea	e9	e8	e7

Contents of memory on the left,

Assume %rax contains 0x10,
%rbx contains 0x40

Specification	Computation	Address	Value
$0x5	<na>	<na>	5
%rax	<na>	<na>	0x10
0x5	0x05	0x05	0xfa
(%rax)	%rax	0x10	0xfe
0x04(%rax)	0x04+%rax	0x14	0xfa
(%rax,%rbx)	%rax+%rbx	0x50	0xfa
0x04(%rax,%rbx)	0x04+%rax+%rbx	0x54	0xf6
(,%rax,4)	%rax*4	0x40	0xfb
0x05 (,%rax,2)	0x05+%rax*2	0x25	0xf8
(%rax,%rbx,2)	%rax+%rbx*2	0x90	0xf6
0x05 (%rax,%rbx,2)	0x05+%rax+%rbx*2	0x95	0xf1

Assembly Language Review

Move instructions: mov source and destination

Combinations of source and destination implied. Proper operation code determined by compiler.

immediate to register	mov	immediate,register
register to register	mov	register,register
memory to register	mov	memory,register
immediate to memory	mov	immediate,memory
register to memory	mov	register,memory

Obviously cannot move to an immediate: mov %rax,$0x40
Cannot move memory to memory

Moving from smaller to larger: sign extension or zero extension

movs	source,dest	sign extension
movz	source,dest	zero extension

Moving from larger to smaller, take only low order.

Move	movb	movw	movl	movq
Move, sign extension	movsbw	movsbl	movswl	movslq
Move, zero extension	movzbw	movzbl	movzwl	movxlq

mov examples

Build up the move instruction:

mov[s|z][l1][l2]

s = sign extension
z = zero extension
l1 = mnemonic of source
l2 = mnemonic of destination

b	byte (8 bits)
w	word (16 bits)
l	long (32 bits)
q	quad (64 bits)
s	single precision (32 bits)
d	double precsions (64 bits)
t	extended precision (128 bits)

s|z omitted if source and destination lengths are the same

movslq mov[s][l][q] = move sign extension, long to quad

mov %rsp,%rbp	register to register	64 bits
movl $0x12345678,-0x4(%rbp)	immediate to memory	32 bits
mov $0x0,%eax	immediate to register	%eax implies 32 bits
mov %rdi,-0x28(%rbp)	register to memory	%rdi implies 64 bits
mov -0x28(%rbp),%rax	memory to register	%rdi implies 64 bits
mov (%rax),%edx	memory to register	%edx implies 32 bits
mov %eax,-0x14(%rbp)	register to memory	%eax implies 32 bits
movzwl -0x14(%rbp),%edx	memory to register	word to long, zero extension
movswl -0x12(%rbp),%eax	memory to register	word to long, sign extension

Assembly Language Review

stack instructions

push	operand	enlarges stack, places operand on stack
	short for	leal -0x08(%rsp),%rsp mov operand,(%rsp)
pop	operand	places top of stack in operand, decreases stack
	short for	mov (%rsp),operand leal 0x08(%rsp),%rsp
call	xyz	enlarges stack, places return address on stack, jumps to xyz

	short for	leal -0x08(%rsp),%rsp
		mov %rip,(%rsp)
		addq $0x05,(%rsp)
		mov address of xyz, %rip

leave		copies base pointer to stack pointer, restores base pointer decrease stack

	short for	mov %rbp,%rsp
		mov (%rbp),%rbp
		leal 0x08(%rsp),%rsp

ret		place return address in rip, decrease stack
	short for	mov (%rsp),%rip leal 0x08(%rsp),%rsp

Assembly Language Review

call/return stack manipulation

```
#include <stdio.h>
void to_sub( int *i )
{
int j = 0x78563412 ; char y[10] ;

int k = *i+j ;
other_sub( &j ) ;
printf( "%p\n", &k ) ;
}

void other_sub( int *j )
{
int i ;
i = *j ;
}

int main()
{
int i; char x[30] ;

i = 0x12345678 ;
to_sub( &i ) ;
return 0 ;
}
```

main

```
4004c4 <+0>:    push   %rbp
4004c5 <+1>:    mov    %rsp,%rbp
4004c8 <+4>:    sub    $0x30,%rsp
4004cc <+8>:    movl   $0x12345678,-0x4(%rbp)
4004d3 <+15>:   lea    -0x4(%rbp),%rax
4004d7 <+19>:   mov    %rax,%rdi
4004da <+22>:   callq  0x4004e6 <to_sub>
4004df <+27>:   mov    $0x0,%eax
4004e4 <+32>:   leaveq
4004e5 <+33>:   retq
```

to_sub

```
4004e6 <+0>:    push   %rbp
4004e7 <+1>:    mov    %rsp,%rbp
4004ea <+4>:    sub    $0x30,%rsp
4004ee <+8>:    mov    %rdi,-0x28(%rbp)
4004f2 <+12>:   movl   $0x78563412,-0x4(%rbp)
4004f9 <+19>:   mov    -0x28(%rbp),%rax
4004fd <+23>:   mov    (%rax),%edx
4004ff <+25>:   mov    -0x4(%rbp),%eax
400502 <+28>:   lea    (%rdx,%rax,1),%eax
400505 <+31>:   mov    %eax,-0x14(%rbp)
400508 <+34>:   lea    -0x4(%rbp),%rax
40050c <+38>:   mov    %rax,%rdi
40050f <+41>:   callq  0x40052f <other_sub>
400514 <+46>:   mov    $0x400648,%eax
400519 <+51>:   lea    -0x14(%rbp),%rdx
40051d <+55>:   mov    %rdx,%rsi
400520 <+58>:   mov    %rax,%rdi
400523 <+61>:   mov    $0x0,%eax
400528 <+66>:   callq  0x4003b8 <printf@plt>
40052d <+71>:   leaveq
40052e <+72>:   retq
```

Assembly Language Review

Before Call

```
Breakpoint 1, 4004da in main ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe548      0x00007fff      0x004005a5      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe420

%rbp → 0x7fffffffe450

i → 0x12345678

rip	rbp	rsp
4da	e450	e420

Assembly Language Review

After Call

```
                  4004e6 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe548      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rbp

rip	rbp	rsp
4e6	e450	e418

Return address, %rsp

executed

```
leal   -0x08(%rsp),%rsp
mov   %rip,(%rsp)
addq  $0x05,(%rsp)
mov   address of xyz, %rip
```

Assembly Language Review

After push %rbp

```
             4004e7 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe410

%rbp → 0x7fffffffe450

(%rbp)

rip	rbp	rsp
4e7	e450	e410

executed

leal -0x08(%rsp),%rsp
mov %rbp,(%rsp)

Assembly Language Review

Mov %rsp,%rbp

%rsp

%rbp

```
                4004ea in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

rip	rbp	rsp
4ea	e410	e410

executed

mov %rsp,%rbp

Assembly Language Review

Sub 0x30,%rsp

```
                  4004ee in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x00000000
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x00000000
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe3e0

%rbp → 0x7fffffffe410

rip	rbp	rsp
4ee	e410	e3e0

executed

sub 0x30,%rsp

Assembly Language Review

Before leave

```
Breakpoint 2, 400528 in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rsp → 0x7fffffffe3e0

%rbp → 0x7fffffffe410

rip	rbp	rsp
528	e410	e3e0

Assembly Language Review

After leave, before ret

```
            40052d in to_sub ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

%rbp

%rsp

rip	rbp	rsp
52d	e450	e418

executed

mov %rbp,%rsp
mov (%rbp),%rbp
leal $0x08(rbp),

Assembly Language Review

After ret

```
                  4004df in main ()
(gdb) x/32w 0x7fffffffe3e0
0x7fffffffe3e0: 0x00000000      0x00000000      0xd08908ed      0x00000032
0x7fffffffe3f0: 0x00000000      0x00000000      0x00400560      0x8a8a8a8a
0x7fffffffe400: 0x00000000      0x00000000      0x004003a3      0x78563412
0x7fffffffe410: 0xffffe450      0x00007fff      0x004004df      0x00000000
%rsp ──────────► 0x7fffffffe420: 0xd080fba0      0x00000032      0x00400560      0x00000000
0x7fffffffe430: 0x00000000      0x00000000      0x004003e0      0x00000000
0x7fffffffe440: 0xffffe530      0x00007fff      0x00000000      0x12345678
%rbp ──────────► 0x7fffffffe450: 0x00000000      0x00000000      0xd081ed1d      0x00000032
```

rip	rbp	rsp
4df	e450	e420

executed

mov (%rsp),%rip
leal 0x08(%rsp),%rsp

Assembly Language Review

Arithmetic and Logical Operations

address	leal	memory,register	load effective address (address arithmetic,destination only a register)
unary	inc	register or memory	increment
	dec	register or memory	decrement
	neg	register or memory	negate
	not	register or memory	complement
arithmetic	add	memory or register,register	add
	sub	memory or register,register	subtract
	imul	memory or register,register	integer multiply
	idiv	memory or register	integer divide (divides RDX:RAX by source)
logical	xor	memory or register,register	bitwise exclusive or
	or	memory or register,register	bitwise or
	and	memory or register,register	bitwise and
shift	sal	immediate or one byte register,memory or register	left arithmetic shift (fill right with zeroes)
	shl	immediate or one byte register,memory or register	left logical shift (sal) (fill right with zeroes)
	sar	immediate or one byte register,memory or register	right arithmetic shift (fill left with sign bit)
	shr	immediate or one byte register,memory or register	right logical shift (fill left with zeroes)

only register %cl allowed for register operand

Practice Problem -- leal

assume %rax contains x, %rbx contains y

	result
leal 6(%rax), %rcx	x+6
leal (%rax,%rbx), %rcx	x+y
leal (%rax,%rbx,4), %rcx	x+4y
leal 7(%rax,%rax,8), %rcx	x+8x+7
leal 0x0a(,%rbx,4), %rcx	10+4y
leal 9(%rax,%rbx,2), %rcx	9+x+2y

Practice Problem -- shifts

assume %rax contains 0x800000000000000f (8 bytes) = 1000 0000 0000 0000 … 0000 0000 0000 1111 (64 bits)

	result	
sal $2,%rax	0x000000000000003c	
shl $2,%rax	0x000000000000003c	
sar $2,%rax	0xe000000000000003	
shr $2,%rax	0x2000000000000003	
sal $63,%rax	0x8000000000000000	
sar $63,%rax	0xFFFFFFFFFFFFFFFF	
leal $0x1,%rax		
sal $63,%rax		
sar $63,%rax	0xFFFFFFFFFFFFFFFF	
and 0xfffffffe,%eax		
sal $63,%rax	0x0000000000000000	// no matter whar is in %rax

using %eax allows only shifts from 0-31 positions (can only shift 0 – length of register -1)

Practice Problem -- convert multiply to shifts

assume x is 10

x * 17	17 = 16 + 1	(x<<4) + x	160 + 10 = 170
x * -7	-7 = -4 -2 -1	-(x<<2) - (x<<1) - x	-40 - 20 - 10 = -70
	also	-(x<<3) + x	-80 + 10
x * 60	60 = 32 + 16 + 8 + 4	(x<<5) + (x<<4) + (x<<3) +(x<<2)	320 + 160 + 80 + 40 = 600
	also	(x<<6) - (x<<2)	640 - 40
x * -112	-112 = -64 – 32 – 16	-(x<<6) - (x<<5) - (x<<4)	-640 - 320 - 160 = -1120
	also	-(x<<7) + (x<<4)	-1280 + 160

What is x<<4 + x ? = x << (4 + x) = x << 14 = 163840 ! shifts have lowest precedence

Assembly Language Review

Practice Problem -- arithmetic

assume	Address	Value	Register	Value
	0x100	0xFF	%eax	0x100
	0x104	0xAB	%ecx	0x1
	0x108	0x13	%edx	0x3
	0x10C	0x11		

Instruction	Destination	Value	
addl %ecx,(%eax)	0x100	0x101	
subl %edx,4(%eax)	0x104	0xA8	
imull $16,(%eax,%edx,4)	0x10c	0x110	0x11 = 0001 0001 * 16 = 0001 0001 0000
incl 8(%eax)	0x108	0x14	
decl %ecx	%ecx	0x0	
subl %edx,%eax	%eax	0xFD	

Assembly Language Review

Status register, set after arithmetic instructions.

Status codes:

logical

CF: Carry Flag. The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations.
(unsigned) t < (unsigned) a when doing t = a+b

ZF: Zero Flag. The most recent operation yielded zero.
t == 0

arithmetic

SF: Sign Flag. The most recent operation yielded a negative value.
t < 0

OF: Overflow Flag. The most recent operation caused a two's-complement overflow—either negative or positive.
(a<0 == b<0) && (t<0 != a<0) when doing t = a+b

Show condtest.c

Compare and test.

all sizes but must be the same. In C, comparing two different sizes, they must first be made the same (larger)size and this depends on signed/unsigned.

arithmetic	cmp	memory or register (S2), memory or register (S1)	set code depending on $S_1 - S_2$
logical	test	memory or register (S2), memory or register (S1)	set code depending on S_1 & S_2
	test	%eax,%eax $S_1 \& S_1 = S_1$! But status is set depending on <,=,> 0.	

set	dest, sets 0 or 1 into dest to store the condition	unsigned and signed suffixes to test for the usual conditions
jmp	label	eg jmpge, jmpne
cmov	memory or register, register conditional move	

suffixes

signed		unsigned		either	
g	>	a	>	e	= 0
ge	>=	ae	>=	ne	!= 0
l	<	b	<	s	< 0
le	<=	be	<=	ns	>= 0

Assembly Language Review

if statements in C

```
if (expression)
  <then statement>
else                 } optional
  <else statement>
```

assembly language:

```
        if ( !expression )
          goto false ;
        then statement
        goto done ;
false:
        else statement
done:
```

```
if ( x<y )
  l = 99 ;
else
  l = 100 ;
```

assembly language:

```
400486 <+18>:   mov    -0xc(%rbp),%eax
400489 <+21>:   cmp    -0x8(%rbp),%eax
40048c <+24>:   jge    0x400497 <main+35>
40048e <+26>:   movl   $0x63,-0xc(%rbp)
400495 <+33>:   jmp    0x40049e <main+42>
false:
400497 <+35>:   movl   $0x64,-0x4(%rbp)
done:
```

Assembly Language Review

do while

```
do {
  statements
   }
while( expression ) ;
```

```
do
  {
  i = i-1 ;    // statements executed at least once
  j = j+1 ;
  }
 while ( i>j ) ;
```

assembly language:

```
loop:
        statements
        if( expression )
          goto loop ;
```

assembly language:

```
loop:
04004cc <+8>:    subl  $0x1,-0x10(%rbp)
04004d0 <+12>:   addl  $0x1,-0xc(%rbp)
04004d4 <+16>:   mov   -0x10(%rbp),%eax
04004d7 <+19>:   cmp   -0xc(%rbp),%eax
04004da <+22>:   jg    0x4004cc <main+8>
04004dc <+24>:
```

Assembly Language Review

while

```
while( expression )
  {
  statements
   }
```

```
while ( i>j )
  {
  i = i-1 ;
  j = j+1 ;
  } ;
```

assembly language:

```
        go to test ;
 loop:
        {
        statements
        }
test:
        if( expression )
           goto loop ;
```

assembly language:

```
4004cc <+8>:    jmp   0x4004d6 <main+18>
loop:
4004ce <+10>:   subl  $0x1,-0x10(%rbp)
4004d2 <+14>:   addl  $0x1,-0xc(%rbp)
test:
4004d6 <+18>:   mov   -0x10(%rbp),%eax
4004d9 <+21>:   cmp   -0xc(%rbp),%eax
4004dc <+24>:   jg    0x4004ce <main+10>
```

Assembly Language Review

for loops

```
for( init; expression, update )
   {
   statements
   }
```

```
for( k=0; k<l; k++ )
    {
    i = i-1 ;
    j = j+1 ;
    }
```

assembly language:

```
                init
                go to test ;
loop:
                {
                statements
                }
                update
test:
                if( expression )
                   goto loop ;
```

assembly language:

```
04004cc <+8>:    movl   $0x0,-0x8(%rbp)  // init
04004d3 <+15>:   jmp    0x4004e1 <main+29>
loop:
04004d5 <+17>:   subl   $0x1,-0x10(%rbp)
04004d9 <+21>:   addl   $0x1,-0xc(%rbp)
04004dd <+25>:   addl   $0x1,-0x8(%rbp)   // update
test:
04004e1 <+29>:   mov    -0x8(%rbp),%eax
04004e4 <+32>:   cmp    -0x4(%rbp),%eax
04004e7 <+35>:   jl     0x4004d5 <main+17>
```

Assembly Language Review

conditional move

var = expression ? true estatement : else statement

m = i < j ? k : l ;

assembly language:

```
t = true statement
if( expression )      implemented
    u = t ;           using cmov
```

assembly language:

```
04004cc <+8>:     mov       -0x10(%rbp),%eax
04004cf <+11>:    cmovge    -0xc(%rbp),%eax
04004d2 <+14>:    mov       %eax,-0x8(%rbp)
```

switch statements

switch (<expression>)
{
case <constant1> :
statements1 // executed when <expression> = <constant1>
// if statements end with **break** ; goto the end
// otherwise, execute the next set of
case <constant2> :
statements2 // executed when <expression> = <constant1>
statements can be null, in which case, the // next
case applies
.
.
default :
default statements
}

Switch statement: Explicit list of cases

```
int switch_eg(int x, int n) {
int result = x;

switch (n) {

  case 100:
  result *= 13;     // x * 13
  break;

  case 102:
  result += 10;    // x + 10
  /* Fall through */

  case 103:
  result += 11;   // x + 11
  break;

  case 104:
  case 106:
  result *= result;  // x * x
  break;

  default:
  result = 0;
  }

return result;
}
```

Case by case outcome?

	result	x=2
<100	0	0
100	x*13	26
101	0	0
102	x+10+11	23
103	x+11	13
104	x*x	4
105	0	0
106	x*x	4
>106	0	0

Assembly Language Review

Assembly with jump table

0x400618	100	0x004004db
	101	0x004004d5
	102	0x004004e2
	103	0x004004e5
	104	0x004004e9
	105	0x004004d5
	106	0x004004e9

```
4004c4 <+0>:    sub    $0x64,%esi
4004c7 <+3>:    cmp    $0x6,%esi
4004ca <+6>:    ja     0x4004d5 <switch_eg+17>
4004cc <+8>:    mov    %esi,%esi
4004ce <+10>:   jmpq   *0x400618(,%rsi,8)
4004d5 <+17>:   mov    $0x0,%eax
4004da <+22>:   retq
4004db <+23>:   lea    (%rdi,%rdi,2),%eax
4004de <+26>:   lea    (%rdi,%rax,4),%eax   // x*13
4004e1 <+29>:   retq
4004e2 <+30>:   add    $0xa,%edi            // x+10
4004e5 <+33>:   lea    0xb(%rdi),%eax       // x+11
4004e8 <+36>:   retq
4004e9 <+37>:   mov    %edi,%eax
4004eb <+39>:   imul   %edi,%eax            // x*x
4004ee <+42>:   retq
```

show switchm1.c, switchm2.c

Assembly Language Review

Procedure calls

Recall stack frame operations:

```
call
push %rbp
sub  0x..,%rsp
leave
ret
```

Registers: caller-save
callee-save

different on x86-64 because more registers

show caller.c with/without –m32

Recursive calls

Stack frame operation ideal: each recursive call has its own context.
N! = factorial defined as $\prod_{i=1}^{n} i$ but also N! = N x (N-1)! And 1! = 1

```
#include <stdio.h>
int factorial( int n ) ;
Int kkk = 0 ;

main()
  {
  int m,n=6 ;
  m = n ;
  printf( "%d factorial is %d\n", m,factorial(n) ) ;
  return 0 ;
  }
int factorial(int n)
  {
  int result ;
  kkk = kkk+1
  if ( n<=1 )
    result = 1 ;
  else
    result = n*factorial(n-1) ;
  printf( "n= %2d kkk= %2d exit factorial result= %d\n", n,kkk,result ) ;
  return result ;
  }
```

Recursive calls

Factorial n stack frame

n = n, returned value

Factorial n-1 stack frame

n = n-1, returned value

.

.

.

Factorial 2 stack frame

n = 2, returned value

Factorial 1 stack frame

n = 1

show factorial.c

Assembly Language Review

Arrays

Arrays: aggregate of same type values:

int x[20] ;   // 20 integer values of x

declarations functions to allocate space and create a pointer

x is a pointer to the start of the array. x[i] is the i_{th} element.

successive memory address of size int: x in memory

0	1	2	.	.	.	.	.	18	19

20 x 4 = 80 bytes total

char y[20] is 20 bytes

int *z[20] is an aray of pointers to integers: 160 bytes total

arithmetic: x+i is the address of element i. Must multiply i by the sizeof(int).

*(x+i) is the value of element i. (dereferencing)

Assembly Language Review

Array arithmetic (int)

```
i : -0x14(%rbp)
x : -0x70(%rbp)
y : -0x10(%rbp)
z : -0x4(%rbp)
```

```c
typedef int T ;
void main() {

int i ;
T x[20] ;
T *y ;
T z  ;

y = x ;
z = x[0] ;
z = x[i] ;
y = &x[2] ;
y = x+i-1 ;
z = *(x+i-3) ;
z = &x[i]-x ;
}
```

```
400478 <+4>:    lea    -0x70(%rbp),%rax
40047c <+8>:    mov    %rax,-0x10(%rbp)          // y = x
400480 <+12>:   mov    -0x70(%rbp),%eax
400483 <+15>:   mov    %eax,-0x4(%rbp)           // z = x[0]
400486 <+18>:   mov    -0x14(%rbp),%eax
40048b <+23>:   mov    -0x70(%rbp,%rax,4),%eax
40048f <+27>:   mov    %eax,-0x4(%rbp)           // z = x[i]
400492 <+30>:   lea    -0x70(%rbp),%rax
400496 <+34>:   add    $0x8,%rax
40049a <+38>:   mov    %rax,-0x10(%rbp)          // y = &x[2]
40049e <+42>:   mov    -0x14(%rbp),%eax
4004a3 <+47>:   sub    $0x1,%rax
4004a7 <+51>:   lea    0x0(,%rax,4),%rdx
4004af <+59>:   lea    -0x70(%rbp),%rax
4004b3 <+63>:   add    %rdx,%rax
4004b6 <+66>:   mov    %rax,-0x10(%rbp)          // y = x+i-1
4004ba <+70>:   mov    -0x14(%rbp),%eax
4004bf <+75>:   sub    $0x3,%rax
4004c3 <+79>:   lea    0x0(,%rax,4),%rdx
4004cb <+87>:   lea    -0x70(%rbp),%rax
4004cf <+91>:   add    %rdx,%rax
4004d2 <+94>:   mov    (%rax),%eax
4004d4 <+96>:   mov    %eax,-0x4(%rbp)           // z = *(x+i+3)
4004d7 <+99>:   mov    -0x14(%rbp),%eax
4004dc <+104>:  shl    $0x2,%rax
4004e0 <+108>:  sar    $0x2,%rax
4004e4 <+112>:  mov    %eax,-0x4(%rbp)           // z = &x[i]-x
```

Assembly Language Review

Array arithmetic (char)

i : -0x14(%rbp)
x : -0x30(%rbp)
y : -0x10(%rbp)
z : -0x1(%rbp)

```
typedef int T ;
void main() {

int i ;
T x[20] ;
T *y ;
T z  ;

y = x ;
z = x[0] ;
z = x[i] ;
y = &x[2] ;
y = x+i-1 ;
z = *(x+i-3) ;
z = &x[i]-x ;
}
```

```
400478 <+4>:   lea    -0x30(%rbp),%rax
40047c <+8>:   mov    %rax,-0x10(%rbp)            // y = x
400480 <+12>:  movzbl -0x30(%rbp),%eax
400484 <+16>:  mov    %al,-0x1(%rbp)              // z = x[0]
400487 <+19>:  mov    -0x14(%rbp),%eax
40048c <+24>:  movzbl -0x30(%rbp,%rax,1),%eax
400491 <+29>:  mov    %al,-0x1(%rbp)              // z = x[i]
400494 <+32>:  lea    -0x30(%rbp),%rax
400498 <+36>:  add    $0x2,%rax
40049c <+40>:  mov    %rax,-0x10(%rbp)            // y = &x[2]
4004a0 <+44>:  mov    -0x14(%rbp),%eax
4004a5 <+49>:  lea    -0x1(%rax),%rdx
4004a9 <+53>:  lea    -0x30(%rbp),%rax
4004ad <+57>:  add    %rdx,%rax
4004b0 <+60>:  mov    %rax,-0x10(%rbp)            // y = x+i-1
4004b4 <+64>:  mov    -0x14(%rbp),%eax
4004b9 <+69>:  lea    -0x3(%rax),%rdx
4004bd <+73>:  lea    -0x30(%rbp),%rax
4004c1 <+77>:  add    %rdx,%rax
4004c4 <+80>:  movzbl (%rax),%eax
4004c7 <+83>:  mov    %al,-0x1(%rbp)              // z = *(x+i-3)
4004ca <+86>:  mov    -0x14(%rbp),%eax
4004cd <+89>:  mov    %al,-0x1(%rbp)              // x = &x[i]-x
```

Structures

Structures: a way of aggregating data into one place. Outcome of declaration is a new data type and creates a map of a contiguous area.

```
struct <new data type name>
  {
  <type> <var1> ;
  <type> <var2> ;
    .
    .
  } <optional name> ;
```

After declaration, <new data type name> is a data type just like char, int, etc.

Variations of Structures

An element can be an array:

```
struct newtype1
  {
   int a ;
   float b[10] ;
   int c ;
   } x ;
```

4 byte int a followed by 10 float b's followed by in c.

Or another structure

```
struct newtype2
{
  int a ;
  struct inner
    {
    float b ;
    int c[10] ;
    } y ;
  int *d ;
} x ;
```

4 byte a followed by a float b, 10 int c's followed by an 8 byte int pointer

Data alignment

Old days: mandatory

Now: instructions work but C aligns in the interest of speed.

malloc always passes back address on 16 byte boundary

Assembly Language Review

Compiled?

```
void main()
  {
  struct newtype1 {
    char a ;
    float b[10] ;
    char c ;
    int d ;
    } x ;
  struct newtype2 {
    char a ;
    double b[10] ;
    char c ;
    int d ;
    } y ;
  struct newtype3 {
    int a ;
    double b[10] ;
    short c ;
    int d ;
    }  z ;

  x.a = 0xff ;
  x.b[0] = 10  ;
  x.c = 0x44 ;
  x.d = 25 ;

  y.a = 0xfe ;
  y.b[0] = 9  ;
  y.c = 0x43 ;
  y.d = 24 ;

  z.a = 0xfd ;
  z.b[0] = 8  ;
  z.c = 0x42 ;
  z.d = 23 ;
  }
```

```
400474 <+0>:     push   %rbp
400475 <+1>:     mov    %rsp,%rbp
400478 <+4>:     sub    $0x88,%rsp
40047f <+11>:    movb   $0xff,-0x40(%rbp)      // x.a      offset -0x40
400483 <+15>:    mov    $0x4120,%eax
400488 <+20>:    mov    %eax,-0x3c(%rbp)       // x.b[0]  offset -0x3c   difference of 4
40048b <+23>:    movb   $0x44,-0x14(%rbp)      // x.c      offset -0x14  difference of 40 = 10 * 4
40048f <+27>:    movl   $0x19,-0x10(%rbp)      // x.d      offset -0x10  difference of 4

400496 <+34>:    movb   $0xfe,-0xa0(%rbp)      // y.a      offset -0xa0
40049d <+41>:    movabs $0x4022,%rax
4004a7 <+51>:    mov    %rax,-0x98(%rbp)       //  y.b[0]  offset -0x98  difference of 8
4004ae <+58>:    movb   $0x43,-0x48(%rbp)      //  y.c     offset -0x48  difference of 80 = 10 * 8
4004b2 <+62>:    movl   $0x18,-0x44(%rbp)      //  y.d     offset -0x44  difference of 4

4004b9 <+69>:    movl   $0xfd,-0x100(%rbp)     // z.a      offset -0x100
4004c3 <+79>:    movabs $0x4020,%rax
4004cd <+89>:    mov    %rax,-0xf8(%rbp)       // z.b[0]   offset -0xf8   difference of 4
4004d4 <+96>:    movw   $0x42,-0xa8(%rbp)      // z.c      offset -0xa8  difference of 80 = 10 * 8
4004dd <+105>:   movl   $0x17,-0xa4(%rbp)      // z.d      offset -0xa4   difference of 4

4004e7 <+115>:   leaveq
4004e8 <+116>:   retq
```

Passing as parameters

```
struct newtype1
                 {
                  int a ;
                  float b[10] ;
                  int c ;
                  } x ;
```

If you want to pass a structure variable as a pointer, you can.

```
void to_sub1( struct newtype1 *x ) ;
```

```
then in to_sub1( struct newtype1 *x )
                              {

                              x->a = 10.5 ;
or
                              (*x).a = 10.5
and
                              (*x).y.b = 11.5
                              }
but x.a no longer works.
```

Compiled code

```
#include <stdio.h>
  struct rect
        {
        int i;
        int j;
        struct inner
          {
          int i ;
          float l ;
          } b ;
        int c[3];
        int *p;
        } x,y ;

  void to_sub1( struct rect x ) ;
  void to_sub2( struct rect *x ) ;


int main()
  {
  int i ;

  printf( "%d\n", sizeof(x) ) ;

  i = 100 ;

  x.i   = 10 ;
  to_sub1( x ) ;
  to_sub2( &x ) ;
  return 0 ;
  }
```

```
Dump of assembler code for function main:
  4004c4 <+0>:      push    %rbp
  4004c5 <+1>:      mov     %rsp,%rbp
  4004c8 <+4>:      sub     $0x40,%rsp
  4004cc <+8>:      mov     $0x4006a8,%eax
  4004d1 <+13>:     mov     $0x28,%esi
  4004d6 <+18>:     mov     %rax,%rdi
  4004d9 <+21>:     mov     $0x0,%eax
  4004de <+26>:     callq   0x4003b8 <printf@plt>
  4004e3 <+31>:     movl    $0x64,-0x4(%rbp)
  4004ea <+38>:     movl    $0xa,0x2004cc(%rip)     # 0x6009c0 <x>
  4004f4 <+48>:     mov     0x2004c5(%rip),%rax     # 0x6009c0 <x>
  4004fb <+55>:     mov     %rax,(%rsp)
  4004ff <+59>:     mov     0x2004c2(%rip),%rax     # 0x6009c8 <x+8>
  400506 <+66>:     mov     %rax,0x8(%rsp)
  40050b <+71>:     mov     0x2004be(%rip),%rax     # 0x6009d0 <x+16>
  400512 <+78>:     mov     %rax,0x10(%rsp)
  400517 <+83>:     mov     0x2004ba(%rip),%rax     # 0x6009d8 <x+24>
  40051e <+90>:     mov     %rax,0x18(%rsp)
  400523 <+95>:     mov     0x2004b6(%rip),%rax     # 0x6009e0 <x+32>
  40052a <+102>:    mov     %rax,0x20(%rsp)
  40052f <+107>:    callq   0x400545 <to_sub1>
  400534 <+112>:    mov     $0x6009c0,%edi
  400539 <+117>:    callq   0x40056c <to_sub2>
  40053e <+122>:    mov     $0x0,%eax
  400543 <+127>:    leaveq
  400544 <+128>:    retq
```

Usage in functions

```
void to_sub1( struct rect x )
  {
  int i ;

  i = 100 ;
  x.i = 10 ;
  x.b.i =  5 ;
  i = x.j   ;
  i = x.b.i ;
  }

void to_sub2( struct rect *x )
  {
  int i ;

  i = 100 ;
  x -> i = 10 ;
  (*x).i = 10 ;

  x -> b.i = 5 ;
  (*x).b.i = 5 ;

  }
```

When passed by name, x.i and x.i.b no longer work because x is a pointer. Must use (*x). instead.

Unions

In unions, the offset is always 0. This means that each variable overlays or occupies the same storage as the other variables:

```
union u
  {
   int i ;
   unsigned char c[4] ;
   float a ;
   } examine_endian ;
```

Sound familiar? Pointers are not needed here! But it is dangerous.

Writing to examine_endian.a overwrites what is in examine_endian.i

Example

```
#include <stdio.h>

void main()
  {
  union endian
        {
        int i ;
        unsigned char c[4] ;
        float a ;
        } hw1 ;
  int j ;

  hw1.i = 19088743   ; /* should be
0x01234567 */

  printf( "\ninteger forward=  " ) ;
  for ( j=0; j<4; j++ ) /* prints c[4] */
        printf( "%02x", hw1.c[j] ) ;
  printf( "\n" ) ;

  printf( "integer backward= " ) ;
  for ( j=3; j>=0; j-- ) /* prints c[4] */
        printf( "%02x", hw1.c[j] ) ;
  printf( "\n\n" ) ;


  hw1.a = 10.01   ;

  printf( "float forward=  " ) ;
  for ( j=0; j<4; j++ ) /* prints c[4] */
        printf( "%02x", hw1.c[j] ) ;
  printf( "\n" ) ;

  printf( "float backward=  " ) ;
  for ( j=3; j>=0; j-- ) /* prints c[4] */
        printf( "%02x", hw1.c[j] ) ;
  printf( "\n\n" ) ;


  }
```

```
Dump of assembler code for function main:
  400554 <+0>:        push    %rbp
  400555 <+1>:        mov     %rsp,%rbp
  400558 <+4>:        sub     $0x10,%rsp
  40055c <+8>:        movl    $0x1234567,-0x10(%rbp)

  400603 <+175>:      mov     $0x412028f6,%eax
  400608 <+180>:      mov     %eax,-0x10(%rbp)
```

```
result of ./a.out

integer forward=    67452301
integer backward= 01234567

float forward=     f6282041
float backward=   412028f6
```

Memory corruption

buffer overflow	gets/puts example
pointer goes wild	pointer error example
array index goes wild	

```
/* Corrupt1.c Stack corruption with gets */
#include <stdio.h>
void echo() ;

int main()
  {
  echo() ;
  printf( "%x\n", EOF ) ;
  }

void echo()
  {
  char inp[8] = "01234567890123456789O" ;

  while ( inp != NULL )
    {
    gets(inp) ;
    puts(inp) ;
    }
  }
```

Out of bounds subscript

```
/* corrupt2.c  Stack corruption with array
overflow */
#include <stdio.h>
void echo() ;
void sub2() ;

int main()
  {
  echo() ;
  printf( "%x\n", EOF ) ;
  }

void echo()
  {
  int i[2] ;
  int j ;
  int k ;

  i[0] = 4 ;
  i[1] = 3 ;
  i[2] = 2 ;
  i[3] = 1 ;
  j = i[4] ;
  i[4] = 0 ; /* destroys return address */
  i[5] = 0 ; /* destroys previous base pointer */
  *(i+4) = -1 ;
  for( k=-4; k>-20; k-- )
     i[k] = k ;

  sub2() ;
  }

void sub2()
  {
  int i = 5 ;
  int j ;

  j = i ;
  }
```

Out of bounds subscript

```
/* corrupt3.c   stack corruption storing outside of frame */
#include <stdio.h>
void echo() ;
void sub2() ;

  int main() {
  echo() ;
  printf( "%x\n", EOF ) ;
  }
void echo() {
  int i[2] ;
  int j ;
  int k ;

  for( k=-4; k>-500; k-- )
    i[k] = k ;

  sub2() ;
  }
void sub2() {
  int i = 5 ;
  int j ;

  j = i ;
  }
```

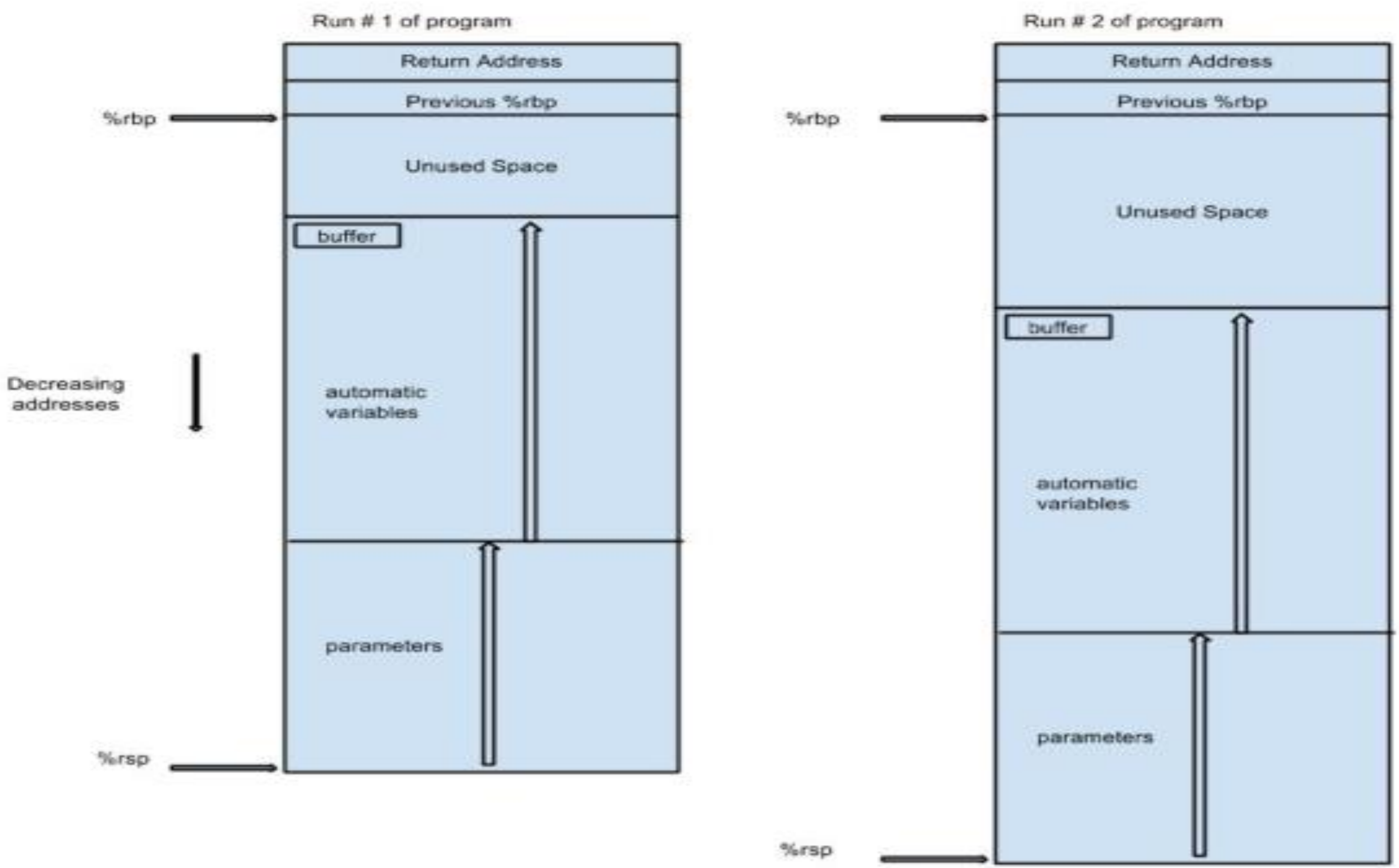
Insert code somewhere (beyond stack). Overlay the return address in the stack. When program returns, it jumps to code.

Stack randomization: after saving the return address and the previous base pointer, allocate a random amount of space in the stack. Then place the automatic variables. This way, the address of the automatic variables and the base pointer has a different offset.

Corruption detection: Store a random value somewhere in stack at the beginning of the program. Store that value in a protected area of memory. At the end of the program compare the values. If changed, raise the red flag.

Hardware which prevents pages from executing code. Memory is divided into 2K or 4K byte “pages”. Each page can be set with read/write/execute bits when in supervisory mode.

Sample Stack Randomization



List of common memory bugs

<u>Passing value instead of pointer</u>:

```
void f1( int *x ) {

  x = 20 ;
  }

void main() {
  int z ;

  f1( z ) ;  // instead if f1( &z ) ;
  }
```

<u>Unitialized values</u>:

```
void main() {
  int i, *a, b[5], c[5] ;

  int *a = (int *)malloc( 5*sizeof(int) ) ;
  for( i=0; i<5; i++ )
    c[i] = a[i]+b[i] ;

  }
```

List of common memory bugs

gets:

```
void f1( int *x ) {
  char b[10] ;

  gets( b ) ;
  }
```

sizeof(int) and sizeof(int *) not the same:

```
void f1( int n ) ;

  int **a = (int **) malloc( n*sizeof(int) ) ;  // instead of sizeof( int * )
 }
```

pointer instead of value referencing:

```
void f1( int *x ) {

*x-- ;  // instead of (*x)-- ;
}
```

List of common memory bugs

<u>pointer arithmetic</u>:

```
void f1( int *x ) {

x += sizeof(int) // instead of x++ ;
}
```

<u>memory pointed to disappears</u>:

```
int *f1( int n ) ;
  int a ;

 return &a ;
 }
```

<u>access freed area</u>:

```
void f1( int *x ) {
int *y ;

y = (int *)malloc( 20*sizeof(int) ) ;
free y ;

for( i=0;i<20; i++ )
   x[i] = y[i] ;
}
```

Optimization Review

Several ways to do it

- The algorithm: smart and mindless ways
  - optimize loops
    - procedure calls
    - recomputing items which do not change
    - unrolling
    - blocking
- The optimizer
- Take advantage of architecture: parallelism, caching

Algorithms: time as a function of set size

- linear or polynomial time
- n^2 (sort1.c)
- n log n (sort2.c)
- accuracy desired? (bin packing)

Algorithms: time as a function of granularity (sort3, sort4)

The importance of measurement

- upper, lower bounds
- average behavior

Speeding up your program

Converting your program from N^2 to N logN: compare the two.

For small N it does not make much difference.

Amdahl's law.

Say T_{old} is the total time a program takes. Let's say part of the program takes a fraction f of the time. Let's speed up that part by a factor of k. then

$$T_{new} = (1-f)*T_{old}+(f*T_{old})/k = T_{old}*((1-f)+f/k)$$

then

$$T_{old} / T_{new} = 1/((1-f)+f/k) = S = \text{ speedup factor}$$

try f = 0.6, k = 3. S = 1.67

Say k is very large, then S is approximately 1/(1-f) and with f = 0.6, S = 2.5

Optimization Review

Compiler vs Programmer

Compiler must analyze code to see where to optimize

- reduce memory references
- take redundant code out of loops
- inline functions

Programmer can do things to allow optimization

- loop unrolling
- taking procedure calls out of loops
- reduce the use of functions: inlining
- reduce memory references
- avoid variable aliasing

Optimization Review

Blocks to optimization

Memory aliasing

```
  void func1(int *xp, int *yp)
  {
1     *xp += *yp;
2     *xp += *yp;
  }

  void func2(int *xp, int *yp)
  {
1     *xp += 2 * *yp;
  }

int main
  {
  int i = 10 ;

  func1( &i, &i ) ;

  i = 10 ;
  func2( &i, &i ) ;

  return 0 ;
  }
```

after line 1 in func1, i = 20, after line 2, i = 40.

after line 1 in func2, i = 30

in both functions, it "looks like" there are two distinct variables but these names are just aliases for the argument.

the compiler could try to optimize func1 to func2 and the compiled code will give different results.

Optimization Review

Blocks to optimization

How about:

```
x = 1000; y = 3000;
*q = y;         /* 3000 */
*p = x;         /* 1000 */
t1 = *q;        /* 1000 or 3000 */
```

if p == q, result is t1 = 1000 ; if not t1 = 3000

Can this happen with pass by value variables?

Blocks to optimization

Consider:

```
void swap(int *xp, int *yp)
  {
  *xp = *xp + *yp; /* x+y */
  *yp = *xp - *yp; /* x+y-y = x */
  *xp = *xp - *yp; /* x+y-x = y */
  }

int main()
  {
  int x,y ;

  x = 10 ;
  y = 20 ;
  swap( &x, &y ) ;

  x = 10 ;
  y = 10 ;
  swap( &x, &y ) ;

  x = 10 ;
  y = 10 ;
  swap( &x, &x ) ;

  return 0 ;
  }
```

```
Start swap
*xp  *yp
 10   20
 30   20
 30   10
 20   10
Start swap
*xp  *yp
 10   10
 20   10
 20   10
 10   10
Start swap
*xp  *yp
 10   10
 20   20
  0    0
  0    0
```

When x != y, (first two cases), everything works normally.
Even when *xp = *yp.

But when x == y, problems.

Blocks to optimization

Consider when a function operates on global variables.

```
int counter = 0;

int f()
  {
  printf( "in f() counter= %d\n", counter ) ;
  return counter++;
  }

int main()
  {
  printf( "result of f()+f()+f()+f() = %d \n", f()+f()+f()+f() ) ;

  counter = 0 ;

  printf( "result of 4*f() = %d \n", 4*f() ) ;

  return 0 ;
  }
```

```
in f() counter= 0
in f() counter= 1
in f() counter= 2
in f() counter= 3
result of f()+f()+f()+f() = 6

in f() counter= 0
result of 4*f() = 0
End value of counter = 1
```

“Optimized” program gives different results.

Measuring performance

Cycles per element

How do we measure performance? Stop watch? Program running time as a function of number of input elements. Also may be a function of the distribution of input data (coarseness).

Run the program many times with different number of input elements and data types. Plot a curve of number of elements versus running time. Fit a line to the data using least squares fit (regression) and you arrive at a formula which is

run time = constant+coefficient * N

where N is the number of input elements. So, the coefficient expresses the rate of increase in run time per additional data element. The constant expresses the overhead to start the program (run time when N = 0).

The coefficient is known as the CPE: cycles per element. Its units are run time per element. In all cases, it is relative to the speed of the computer but we think of it as cycles

Loop unrolling

```
/* Compute prefix sum of vector a */
void psum1(float a[], float p[], long int n)
{
long int i;
p[0] = a[0];
for (i = 1; i < n; i++)
  p[i] = p[i-1] + a[i];
}
```

This function computes the "prefix sum" of an array of elements: a. It is defined as:

p[0] = a[0] ;

p[i] = p[i-1]+a[i]

So, p[i] = the sum of all a[j] where j <= i

```
void psum2(float a[], float p[], long int n)
{
long int i;
p[0] = a[0];
for (i = 1; i < n-1; i+=2) {
  p[i] =  p[i-1] + a[i];
  p[i+1] = p[i] + a[i+1];
  }
/* For odd n, finish remaining element */
if (i < n)
  p[i] = p[i-1] + a[i];
  }
```

This is 1x unrolling

Loop unrolling

*Caveat** The increase in the number of lines in the code affects the savings for loop unrolling.

Lets say that it takes x units of time to execute the line of code in the loop in psum1, y units to execute the loop overhead and z time units are used when the loop is unrolled in psum2.

So, the time to execute the loop in psum1 is

a = x*n+y*n   execute the code plus the loop overhead

an unrolled loop program would take

b = z*n/2+y*n/2   execute the code half as much and the loop overhead half as much

for it to be faster, we want b < a or

z*n/2+y*n/2 < x*n+y*n

This is the same as

0 < x*n+y*n/2-z*n/2

dividing by n it becomes

0 < x+y/2-z/2 = c = difference in run times old - new

Optimization Review

Loop unrolling

So, depending on the relative values of x, y and z, there is a diminishing rate of return!

x	y	z	c
1	1	2	0.5
1	1	3	0
1	1	4	-0.5
1	1	5	-1
2	1	3	1
2	1	4	0.5
2	1	5	0
2	1	6	-0.5

For the first line in the table, we increase the statement executions by 1 and we get a .5 improvement in the run time, by 2 and we get zero. In line 5, we increase it by 1 from 2 to 3, we get an improvement of 1.