CS 110
Computer Architecture
Lecture 24:
Review for Midterm

Instructors:
Sören Schwertfeger & Chundong Wang

https://robotics.shanghaitech.edu.cn/courses/ca/20s/

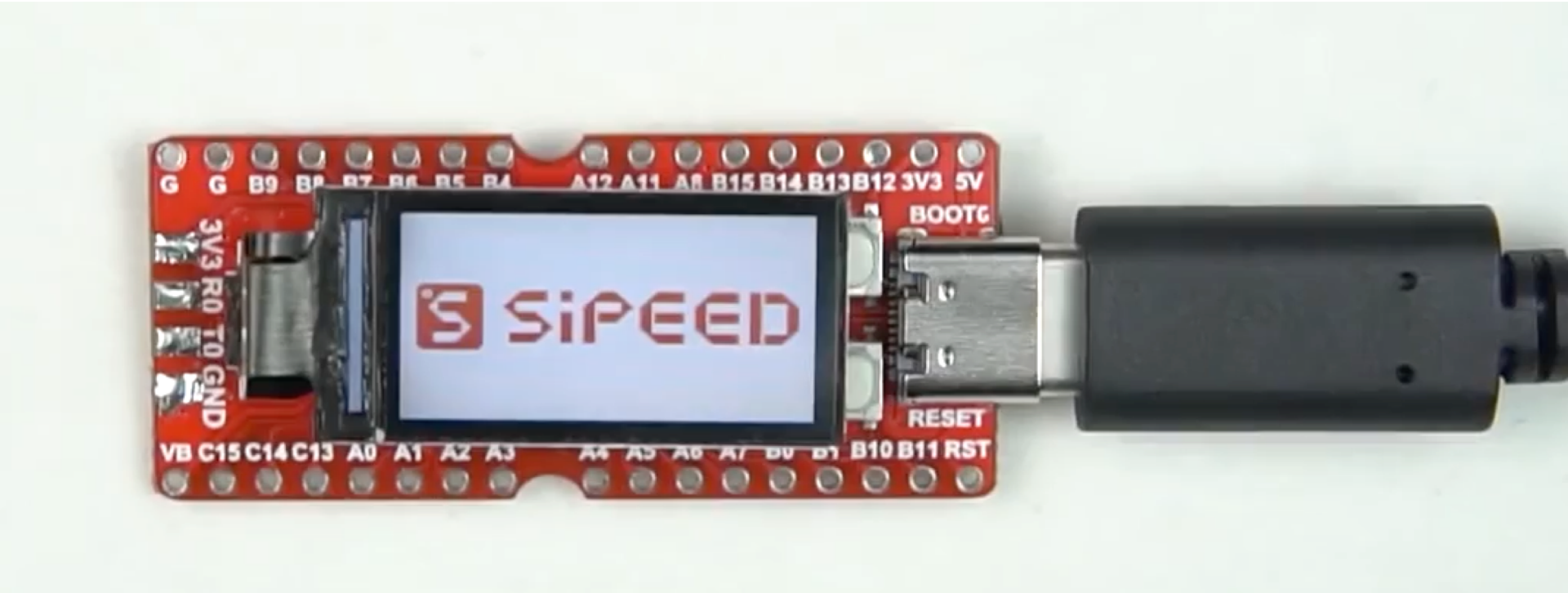
School of Information Science and Technology SIST

ShanghaiTech University

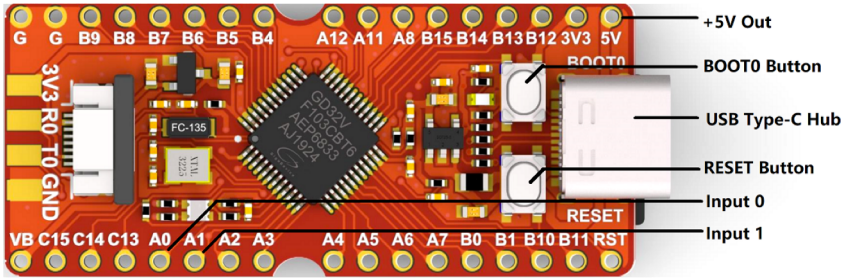
Slides based on UC Berkley's CS61C

Project 4

- Program on a RISC-V CPU!
- Sipeed Longan Nano Development Board with
  - RISC-V 32bit CPU!
  - Comes with a Screen!

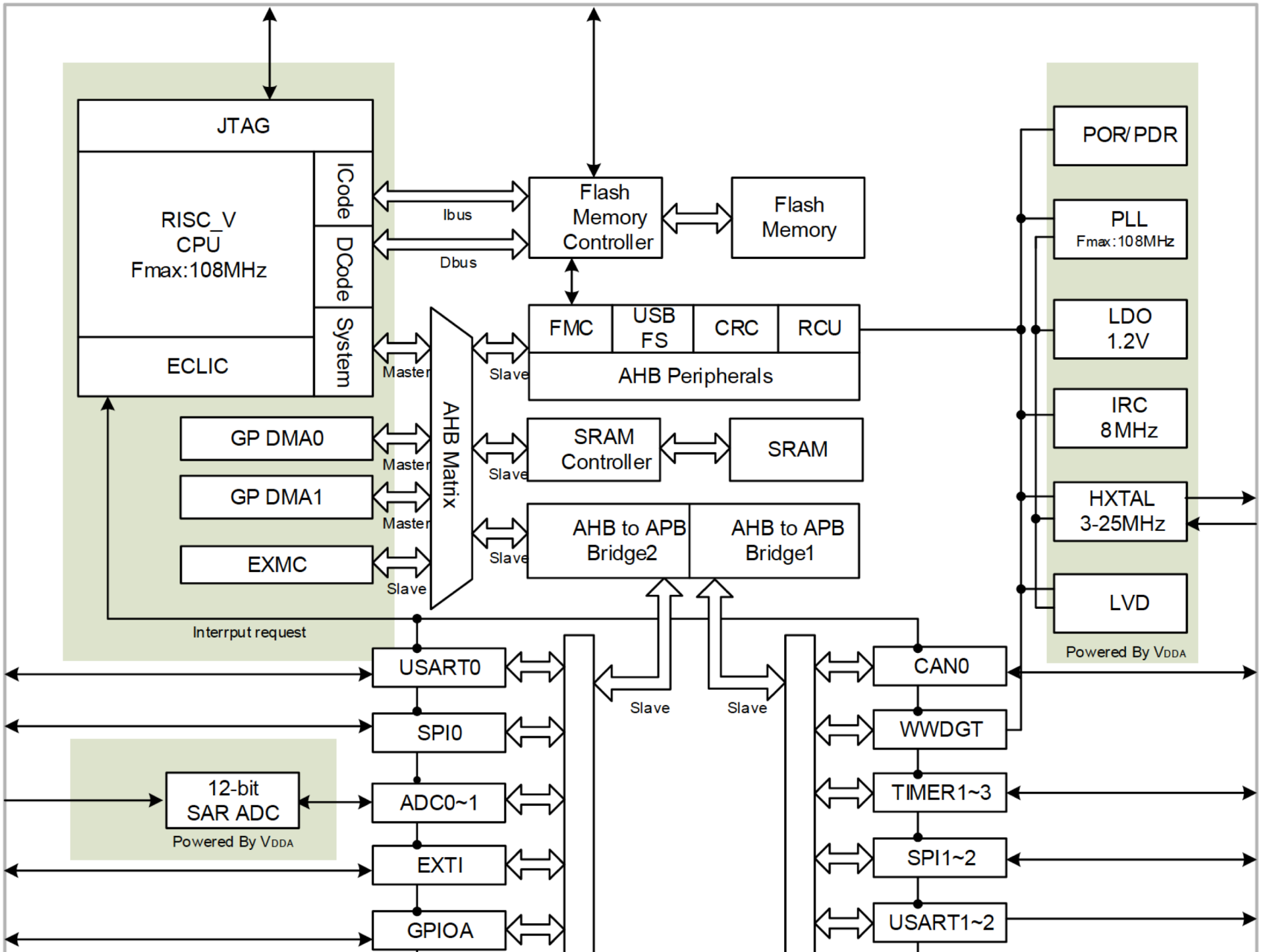


32 bit RISC-V CPU



- Gigadevince GD32VF103CBT6 GD32VF103 “Bumblebee Core”
- ISA: RV32IMAC
  - Integer; Multiplication & Division; Atomic Operations; Compressed (16bit) instructions
  - Single-cycle hardware multiplier and Multi-cycles hardware divider
- 108MHz
- 128kb Flash
- 32kb SRAM
- Cache?
  - No Cache needed: 0 wait states for Flash and SRAM!
    (CPU speed is low while using modern memory)
- Supports misaligned memory access operations (Load/Store)
- Virtual Memory?
  - No Virtual Memory – everything (Flash, SRAM, devices) is mapped to the 32bit address space.

- DMA?
  - Yes!
  - peripheral to memory, memory to peripheral, memory to memory DMA modes
- Interrupts?
  - Yes!
  - Supports the RISC-V architecturally defined software, timer and external interrupts.
  - Dozens of external interrupt sources
  - Programmable 16 interrupt levels and priorities
- Power Saving Modes?
  - Yes!
  - Sleep (core clock off – interrupts run)
  - Deep-Sleep (most interrupts off)
  - Standby (SRAM and registers are lost – save state to flash! Few interrupts available)
- Pipelining?
  - 2-stage pipeline!
- Simple dynamic branch predictor
- Instruction fetch unit (IFU) can prefetch the following two instructions to mask the instruction memory access latency
- Support Machine Mode and User Mode


JTAG
RISC_V
CPU
Fmax:108MHz
ICode
DCode
System
ECLIC
Ibus
Dbus
Flash
Memory
Controller
Flash
Memory
FMC
USB
FS
CRC
RCU
AHB Peripherals
Master
Slave
AHB Matrix
GP DMA0
GP DMA1
EXMC
SRAM
Controller
SRAM
AHB to APB
Bridge2
AHB to APB
Bridge1
Interrput request
POR/PDR
PLL
Fmax:108MHz
LDO
1.2V
IRC
8MHz
HXTAL
3-25MHz
LVD
Powered By VDDA
USART0
SPI0
12-bit
SAR ADC
Powered By VDDA
ADC0~1
EXTI
GPIOA
CAN0
WWDGT
TIMER1~3
SPI1~2
USART1~2

Memory Map

Pre-defined Regions	Bus	Address	Peripherals
External device	AHB	0xA000 0000 - 0xA000 0FFF	EXMC - SWREG
External RAM	AHB	0x9000 0000 - 0x9FFF FFFF	Reserved
	AHB	0x7000 0000 - 0x8FFF FFFF	Reserved
	AHB	0x6000 0000 - 0x6FFF FFFF	EXMC - NOR/PSRAM/SRAM
Peripheral	AHB	0x5000 0000 - 0x5003 FFFF	USBFS
	AHB	0x4008 0000 - 0x4FFF FFFF	Reserved
	AHB	0x4002 3C00 - 0x4002 3FFF	Reserved
	AHB	0x4002 3800 - 0x4002 3BFF	Reserved
	AHB	0x4002 3400 - 0x4002 37FF	Reserved
	AHB	0x4002 3000 - 0x4002 33FF	CRC
	APB2	0x4001 3C00 - 0x4001 3FFF	Reserved
	APB2	0x4001 3800 - 0x4001 3BFF	USART0
	APB2	0x4001 3400 - 0x4001 37FF	Reserved
	APB2	0x4001 3000 - 0x4001 33FF	SPI0
	APB2	0x4001 2C00 - 0x4001 2FFF	TIMER0
	APB2	0x4001 2800 - 0x4001 2BFF	ADC1
	APB2	0x4001 2400 - 0x4001 27FF	ADC0
	APB2	0x4001 2000 - 0x4001 23FF	Reserved
	APB2	0x4001 1C00 - 0x4001 1FFF	Reserved
	APB2	0x4001 1800 - 0x4001 1BFF	GPIOE
	APB2	0x4001 1400 - 0x4001 17FF	GPIOD
	APB2	0x4001 1000 - 0x4001 13FF	GPIOC
	APB2	0x4001 0C00 - 0x4001 0FFF	GPIOB
	APB2	0x4001 0800 - 0x4001 0BFF	GPIOA
	APB2	0x4001 0400 - 0x4001 07FF	EXTI
	APB2	0x4001 0000 - 0x4001 03FF	AFIO
	APB1	0x4000 7C00 - 0x4000 7FFF	Reserved
	APB1	0x4000 7800 - 0x4000 7BFF	Reserved
	APB1	0x4000 7400 - 0x4000 77FF	DAC
	APB1	0x4000 7000 - 0x4000 73FF	PMU
	APB1	0x4000 6C00 - 0x4000 6FFF	BKP
	APB1	0x4000 6800 - 0x4000 6BFF	CAN1
	APB1	0x4000 6400 - 0x4000 67FF	CAN0
	APB1	0x4000 6000 - 0x4000 63FF	Shared USB/CAN SRAM 512 bytes
	APB1	0x4000 5C00 - 0x4000 5FFF	USB device FS registers
	APB1	0x4000 5800 - 0x4000 5BFF	I2C1
	APB1	0x4000 5400 - 0x4000 57FF	I2C0
	APB1	0x4000 5000 - 0x4000 53FF	UART4
	APB1	0x4000 4C00 - 0x4000 4FFF	UART3
	APB1	0x4000 4800 - 0x4000 4BFF	USART2
	APB1	0x4000 4400 - 0x4000 47FF	USART1
	APB1	0x4000 4000 - 0x4000 43FF	Reserved
	APB1	0x4000 3C00 - 0x4000 3FFF	SPI2/I2S2
	APB1	0x4000 3800 - 0x4000 3BFF	SPI1/I2S1
	APB1	0x4000 3400 - 0x4000 37FF	Reserved
	APB1	0x4000 3000 - 0x4000 33FF	FWDGT
	APB1	0x4000 2C00 - 0x4000 2FFF	WWDGT
	APB1	0x4000 2800 - 0x4000 2BFF	RTC
	APB1	0x4000 2400 - 0x4000 27FF	Reserved
	APB1	0x4000 2000 - 0x4000 23FF	Reserved
	APB1	0x4000 1C00 - 0x4000 1FFF	Reserved
	APB1	0x4000 1800 - 0x4000 1BFF	Reserved
	APB1	0x4000 1400 - 0x4000 17FF	TIMER6
	APB1	0x4000 1000 - 0x4000 13FF	TIMER5
	APB1	0x4000 0C00 - 0x4000 0FFF	TIMER4
	APB1	0x4000 0800 - 0x4000 0BFF	TIMER3
	APB1	0x4000 0400 - 0x4000 07FF	TIMER2
	APB1	0x4000 0000 - 0x4000 03FF	TIMER1
SRAM	AHB	0x2007 0000 - 0x3FFF FFFF	Reserved
	AHB	0x2006 0000 - 0x2006 FFFF	Reserved
	AHB	0x2003 0000 - 0x2005 FFFF	Reserved
	AHB	0x2002 0000 - 0x2002 FFFF	Reserved
	AHB	0x2001 C000 - 0x2001 FFFF	Reserved
	AHB	0x2001 8000 - 0x2001 BFFF	Reserved
	AHB	0x2000 5000 - 0x2001 7FFF	SRAM
	AHB	0x2000 0000 - 0x2000 4FFF	
Code	AHB	0x1FFF F810 - 0x1FFF FFFF	Reserved
	AHB	0x1FFF F800 - 0x1FFF F80F	Option Bytes
	AHB	0x1FFF B000 - 0x1FFF F7FF	Boot loader
	AHB	0x1FFF 7A10 - 0x1FFF AFFF	Reserved
	AHB	0x1FFF 7800 - 0x1FFF 7A0F	Reserved
	AHB	0x1FFF 0000 - 0x1FFF 77FF	Reserved
	AHB	0x1FFE C010 - 0x1FFE FFFF	Reserved
	AHB	0x1FFE C000 - 0x1FFE C00F	Reserved
	AHB	0x1001 0000 - 0x1FFE BFFF	Reserved
	AHB	0x1000 0000 - 0x1000 FFFF	Reserved
	AHB	0x083C 0000 - 0x0FFF FFFF	Reserved
	AHB	0x0802 0000 - 0x083B FFFF	Reserved
	AHB	0x0800 0000 - 0x0801 FFFF	Main Flash
	AHB	0x0030 0000 - 0x07FF FFFF	Reserved
	AHB	0x0010 0000 - 0x002F FFFF	Aliased to Main Flash or Boot loader
	AHB	0x0002 0000 - 0x000F FFFF	
	AHB	0x0000 0000 - 0x0001 FFFF	

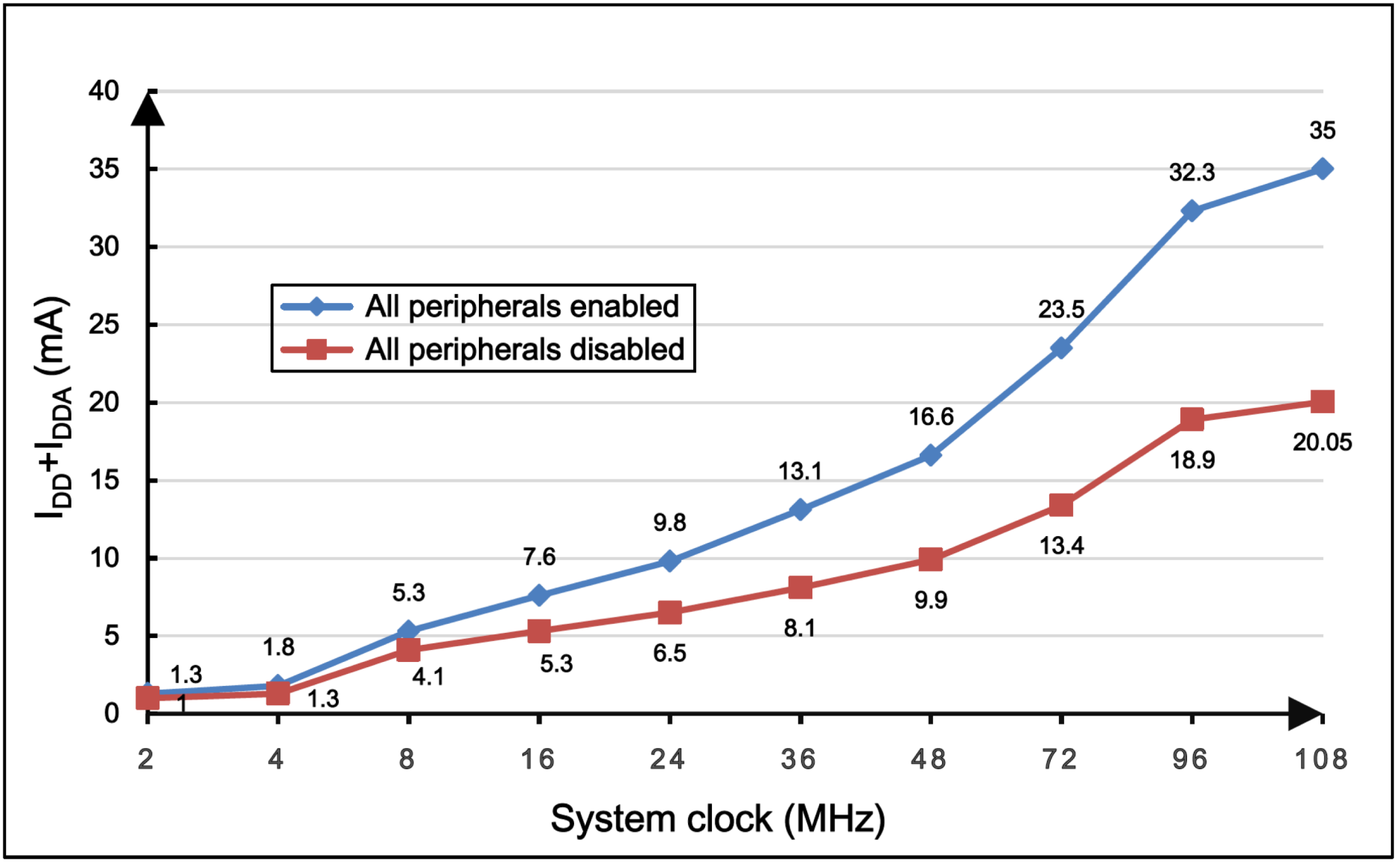
2.3.1. Flash memory architecture

The flash memory consists of up to 128 KB main flash organized into 128 pages with 1 KB capacity per page and a 18 KB Information Block for the Boot Loader. The main flash memory contains a total of up to 128 pages which can be erased individually. The ***Table 2-1. Base address and size for flash memory*** shows the details of flash organization.

Table 2-1. Base address and size for flash memory

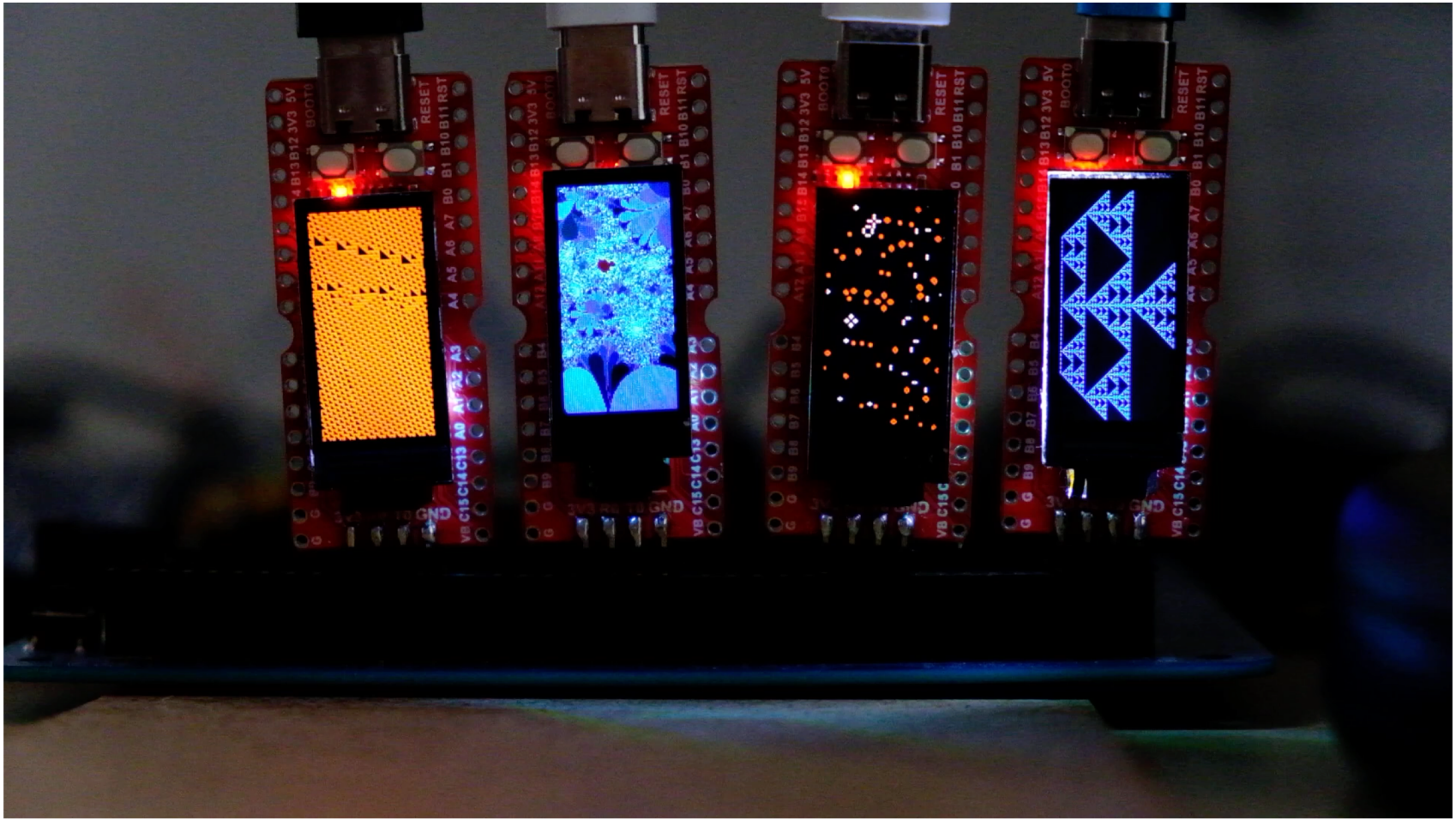
Block	Name	Address Range	size (bytes)
Main Flash Block	Page 0	0x0800 0000 - 0x0800 03FF	1KB
	Page 1	0x0800 0400 - 0x0800 07FF	1KB
	Page 2	0x0800 0800 - 0x0800 0BFF	1KB
	…	…	…
	Page 127	0x0801 FC00 - 0x0801 FFFF	1KB
Information Block	Boot loader area	0x1FFF B000- 0x1FFF F7FF	18KB
Option bytes Block	Option bytes	0x1FFF F800 - 0x1FFF F80F	16B

Figure 4-2. Typical supply current consumption in Run mode



- V_{dd} = 3.3V => 35mA @108MHz => 0.12W

Cool Projects



Project 4

- Program in C *and* RISC-V on the Logan Nano
- Implement Pong Game
  - OR some better game if your want
- Call C functions from RISC-V!
  - Good – we do NOT need to program everything by hand on hardware – use provided C library!
- Lab 11: get familiar with the Logan Nano
- Lab 14 in week 18 ("2nd final week"): Demo and checkup of Project 4

We provide each project group with:

- 1 x Sipeed Logan Nano with screen and housing RMB 34.8 https://item.taobao.com/item.htm?id=601743142093
- 2 x push buttons https://detail.tmall.com/item.htm?id=554574318222

- **<u>You need to provide your own USB-C cable!</u>**
- You are free to buy your own hardware (e.g. your own Logan Nano, buttons, potentiometers, speaker!?) for Project 4…
- Code still goes to gitlab!

Links:

- https://longan.sipeed.com/en/
- https://www.gigadevice.com/microcontroller/gd32vf103cbt6/
- https://github.com/nucleisys/Bumblebee_Core_Doc
- https://docs.platformio.org/en/latest/platforms/gd32v.html

- Lab 11: https://robotics.shanghaitech.edu.cn/courses/ca/20s/labs/11/

Midterm

- Date: Tuesday, May. 26
- Time: 10:15- 12:15 (normal lecture slot++)
  - Be there latest 10:00 – we start 10:15 sharp!
- Venue: 4 rooms – check on egate which room you are!:
  - SPST1-503
    SPST1-201
    SPST1-501
    SIST1A-106
- Closed book:
  - You can bring **two** A4 pages with notes (both sides; in **English**): Write your Chinese and **Pinyin** name on the top! **Handwritten** by you!
    - **Final**: you can bring **three** A4 pages
  - You will be provided with the RISC-V ”green sheet”
  - No other material allowed!

Midterm I

- Wear your Corona mask! =>
- Switch cell phones **off**!
  (not silent mode – off!)
  - Put them in your bags.
- Bags under the table. Nothing except paper, pen, 1 drink, 1 snack, your student ID card on the table!
- No other electronic devices are allowed!
  - No ear plugs, music, smartwatch…
- Anybody touching any electronic device will **FAIL** the course!
- Anybody found cheating (copy your neighbors answers, additional material, ...) will **FAIL** the course!


CASIO
JF-120TM
TWO WAY POWER
12 DIGITS
F CUT 5/4
4 3 2 1 0 A
+/-
MRC
TAX-
TAX RATE
RATE SET
%
TAX+
GT
C
ON
AC
7
8
9
4
5
6
1
2
3
0
00
.
=
+
-
×


08:08
Thu, Sep 1
HUAWEI


Tue
9


COMPUTER
ORGANIZATION
AND DESIGN
THE HA WARE/SOFTWA INTERFACE
RISC-V EDITION
MK
MORGAN KAUFMANN
DAVID A. PATTERSON
JOHN L. HENNESSY

FUNCTIONS OF SEVERAL VARIABLES

LEVEL CURVES

FUNCTION OF n VARIABLES

PARTIAL DERIVATIVES

SECOND PARTIAL DERIVATIVES

CLAIRAUT'S THEOREM

THE CHAIN RULE

IMPLICIT DIFFERENTIATION

TANGENT PLANE TO A LEVEL SURFACE

NORMAL LINE TO A LEVEL SURFACE

MAXIMUM AND MINIMUM VALUES

FINDING ABSOLUTE MAX. AND MINS.

MAXIMIZING AND MINIMIZING

DOMAINS: Allowed (x,y), (x,y,z) RANGES: 2D's

ε-δ DEFINITION OF CONTINUITY

COMPOSITE FUNCTIONS OF CONTINUOUS FUNCTIONS ARE CONTINUOUS, AS ARE SUMS AND PRODUCTS

EQUATIONS OF TANGENT PLANES TO SURFACES

TOTAL DIFFERENTIAL

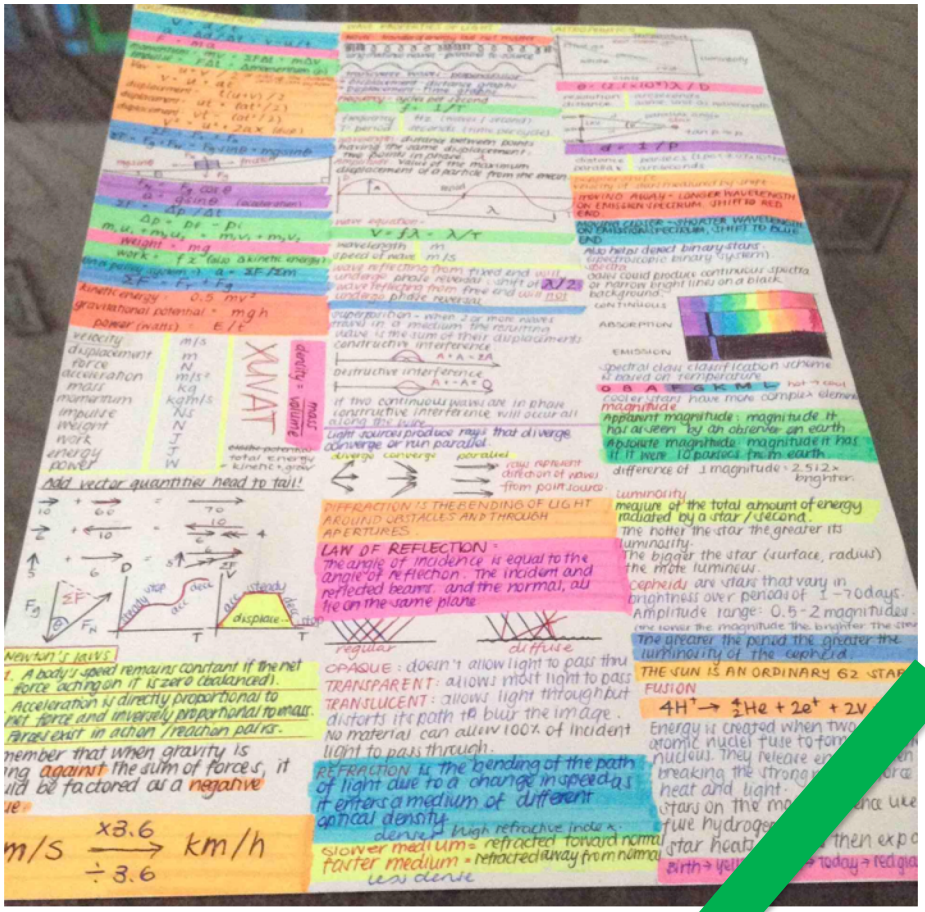
INCREMENTS $\Delta x, \Delta y, \Delta z$ DIFFERENTIALS dx, dy, dz

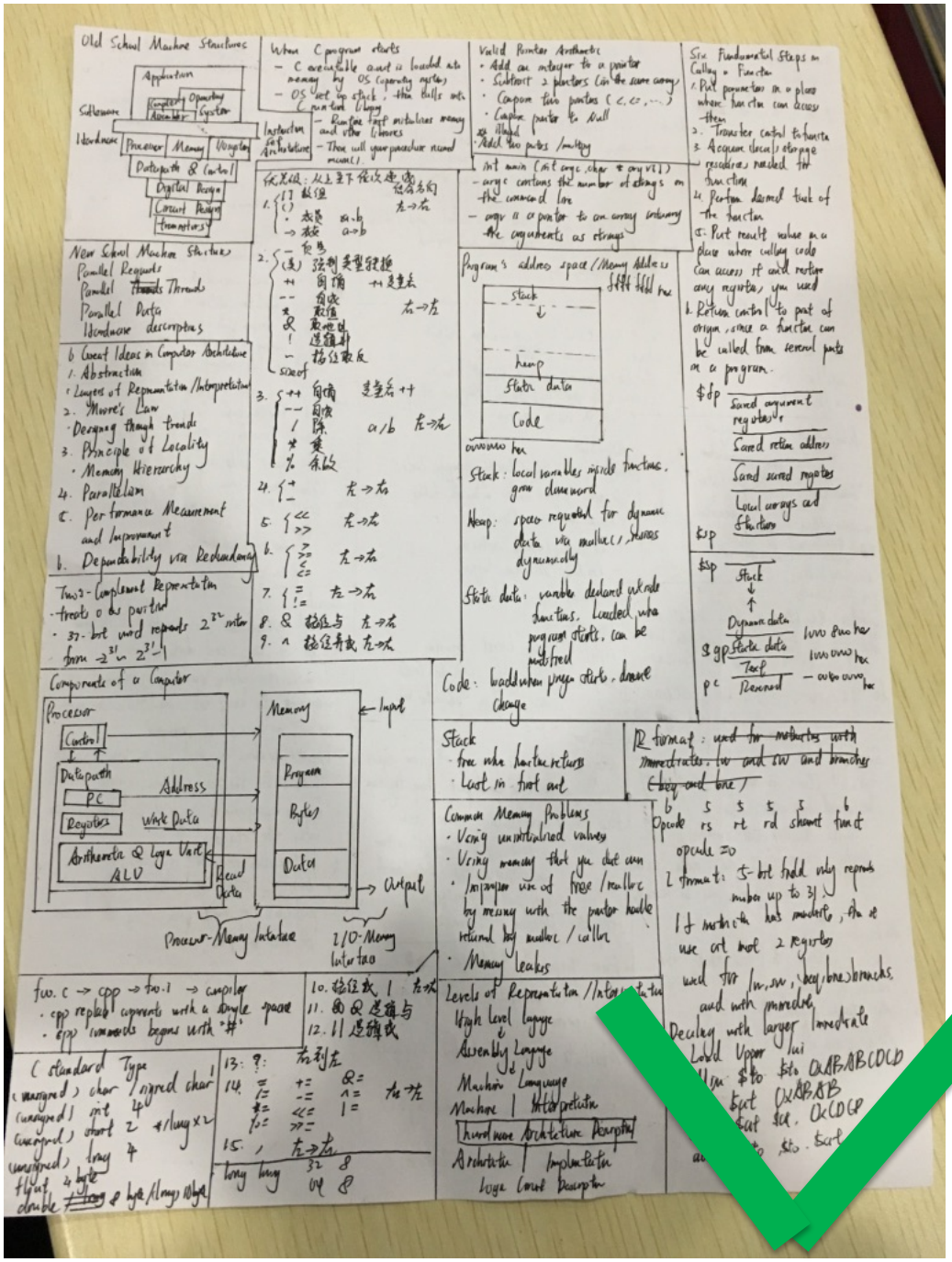
THEOREM

DEPENDENCY DIAGRAMS

THE GRADIENT VECTOR

DIRECTIONAL DERIVATIVES




Old School Machine Structures
Application
Operating System
Compiler
Assembler
Hardware
Processor
Memory
I/O
Datapath & Control
Digital Design
Circuit Design
Transistors
New School Machine Structures
Parallel Requests
Parallel Threads
Parallel Data
Hardware descriptions
6 Great Ideas in Computer Architecture
1. Abstraction
2. Moore's Law
3. Principle of Locality
4. Parallelism
5. Performance Measurement and Improvement
6. Dependability via Redundancy
Two's-Complement Representation
Components of a Computer
Processor
Control
Datapath
PC
Registers
Arithmetic & Logic Unit
ALU
Address
Write Data
Read Data
Memory
Program
Bytes
Data
Input
Output
Processor-Memory Interface
I/O-Memory Interfaces
Valid Pointer Arithmetic
Add an integer to a pointer
Subtract 2 pointers (in the same array)
Compare two pointers (<, <=, ...)
Compare pointer to NULL
Stack
Heap
Static data
Code
Common Memory Problems
Using uninitialized values
Using memory that you don't own
Improper use of free/realloc by messing with the pointer handle returned by malloc/calloc
Memory leaks
Levels of Representation/Interpretation
High Level Language
Assembly Language
Machine Language
Machine Interpretation
Hardware Architecture Description
Architecture Implementation
Logic Circuit Description
Standard Types
char
int
short
long
float
double
long long
Stack
free in hardware
Last in first out
Six Fundamental Steps in Calling a Function
Saved argument registers
Saved return address
Saved saved registers
Local arrays and structures
$sp
$gp
pc
Dynamic data
Static data
Text
Reserved
Opcode
rs
rt
rd
shamt
funct
I format
J format
Decoding with larger immediate
Load Upper Imi

SIFT REFERENCE GUIDE (V.1.1) – CREATING TIMELINES WITH THE SIFT WORKSTATION

SANS COMPUTER FORENSICS and INCIDENT RESPONSE

THE PURPOSE OF THIS REFERENCE GUIDE IS TO WALK THROUGH THE PROCESS OF BOOTING THE SIFT WORKSTATION, CREATING A TIMELINE ("SUPER" OR "MICRO") AND REVIEWING IT.

1. VISIT: http://computer-forensics11.sans.org/community/downloads

- **oad:** SIFT Workstation VM Appliance
- **Do** FT Workstation Installation

2. BOOT SIFT VM

Login: *sansforensics*
Password: *forensics*

3. ELEVATE PRIVS

$ sudo su

4. CONNECT IMAGE TO SIFT

Plug hard drive to physical host and attach to SIFT VM

5. HARD DRIVE MOUNTING *(if you are using log2timeline-sift and Single DD you can skip to 7-A)*

SINGLE OR SPLIT IMAGE (2 options):

EWF/

- **nt_ewf.py image.E01 /mnt/ewf**
- **# ... image.E01 /mnt/ewf/**

Not Needed For 7-A

 # mount -t ntfs -o ro,loop,show_sys_files,streams_interface=wind... offset=#### /mnt/ewf/<image> /mnt/windows_mount/

MOUNT TO MOUNT POINT

DD

SINGLE IM...

 # mount -t ntfs -o ... w_sys_files,streams_interface=windows,offset=#### image.dd /m... ows_mount/

SPLIT IMAGE (2 step proc...

 # affuse image.001 /mnt/aff
 # mount –t ntfs-3g –o loop,ro,show_... /mnt/aff/<image> /mnt/windows_...

HOW TO CALCULATE THE OFFSET FOR MOUNTING

1. Run mmls to query partition layout
 # mmls image.E01
2. Identify partition and byte offset
3. (Partition byte offset) x (bytes per sector) = **offset #### to use!**
Example: 63 X 512 = 32256

Note: If needed, repeat for each partition. Make new mount point:
***# mkdir** /mnt/windows_mount2/*

6. log2timeline default timezone is set to examiner local host. To change use -z [TIMEZONE] option. To list all available timezones:
 # log2timeline -z list

7-A & 7-B

7-A: AUTOMATED SUPER TIMELINE CREATION

log2timeline-sift -o –z [TIMEZONE] -p [PARTITION #] -i [IMAGE FILE]

DISK IMAGE (prompt for partition, mount, and run):

XP **# log2timeline-sift –z EST5EDT -i image**

WIN7 **# log2timeline-sift -win7 -z EST5EDT -i image**

FOR PARTITION (mount and run using all applicable pl...

XP **# log2timeline-sift –z EST5EDT -p 0 -i parti...**

WIN7 **# log2timeline-sift -win7 -z EST5EDT ...tion**

OTHER USAGE EXAMPLES:

Display list of available plu...
 # log2timeline -f list
Run log2timeline use... use only specific plugins:
 # log2timeline-sift ...refetch –z EST5EDT -i image.dd
Help (man page...
 # log2timeli...

...NUAL "MICRO" TIMELINE CREATION

...TIONS] [-f FORMAT] [-z TIMEZONE] [-o OUTPUT MODULE] [-w ...G_FILE/LOG_DIR [--] [FORMAT FILE OPTIONS]

...EM METADATA (using log2timeline or fls)

...stem data w/log2timeline from mounted file system:
 # log... ...f mft -o mactime –r -z EST5EDT -w
mft.bo... ...olume/
OR Extract M... ...age using Sleuthkit:
 # fls -m "" -o offs... ...dd > fls.body
Convert body file for... ...format w/ mactime:
 # mactime –b fls.body –d...

ARTIFACTS (run l2l on mounted file s... ...h plugins recursively)

Extract artifacts w/ log2timeline and ru... ...d file system:
 # log2timeline -f firefox3,chrome -o mactim... ...EDT -w
web.body /mnt/volume/
Convert body file format to CSV format w/ mactime:
 # mactime –b log2timeline.body –d > log2timeline.csv

HELP? OPTIONS? USAGE?

log2timeline -help
Log2timeline-sift -help
L2t_process -help

OTHER log2timeline OUTPUT FORMATS

Note: CSV is Default Output

-BeeDocs - Mac OS X visualization tool
-CEF - Common Event Format - ArcSight
-CFTL - XML file- CyberForensics TimeLab visualization tool
-CSV - comma separated value file
-Mactime - Both older and newer version of the format supported for use by TSK's mactime
-SIMILE - XML file - SIMILE timeline visualization widget
-SQLite - SQLite database
-TLN - Tab Delimited File
-TLN - Format used by some of H Carvey tools, expressed as a ASCII output
-TLNX - Format used by some of H Carvey tools, expressed as a XML document

8. CSV F... ...PUT *(/cases/timeline-output-folder)*

-dat... ...e event, in the format of MM/DD/YYYY
-t... ...of day, expressed in a 24h format, HH:MM:SS
-z: the timezone that was used to call the tool with.
-m: MACB meaning of the fields, comp w/ mactime format.
-source: Source short name (i.e. registry entries are REG)
-sourcetype: Desc of the source ("Internet Explorer" instead of WEBHIST)
-type: Timestamp type (i.e. "Last Accessed", "Last Written")
-user: Username associated with the entry, if one is available.
-host: Hostname associated with the entry, if one is available.
-short: Contains less text than the full description field.
-desc: where majority info is stored, the actual parsed desc of the entry.
-version: Version number of the timestamp object.
-filename: Filename with the full path that contained the entry
-inode: inode number of the file being parsed.
-notes: Some input modules insert additional information in the form of a note, which comes here. Or it can be used during the review.
-format: Input module name used to parse the file.
-extra: Additional information parsed is joined together and put here.

9. FILTER TIMELINE

Filter timeline with date range to include only:
l2t_process -b timeline.csv MM-DD-YYYY..MM-DD-YYYY > filtered.csv
Filter timeline with keyword list (one term per line in keywords.txt):
l2t_process -b timeline.csv -k keywords.txt > filtered.csv
What sources are in your timeline?
awk–F , '{print $6;}' timeline.csv| grep–v sourcetype|sort | uniq
Find all LNK files that reference E Drive
grep"Shortcut LNK" timeline.csv| grep"E:"
FiindMountPoints2 entries that reference E Drive
grep"MountPoints2 key" timeline.csv} | grep"E drive"
grepUSB timeline.csv| grep"SetupAPILog"

File System	M	A	C	B
Ext2/3	Modified	Accessed	Changed	N/A
FAT	Written	Accessed	N/A	Created
NTFS	File Modified	Accessed	MFT Modified	Created
UFS	Modified	Accessed	Changed	N/A

10. CONNECT TO SIFT

✓ ... > SETTINGS -> OPTIONS -> Shared Folders -> Always Enabled (Check)
✓ 2. SIFT Desktop > VMware-Shared-Drive
✓ Access from a Win Machine \\SIFTWORKSTATION

11. REVIEW TIMELINE

Review timelines using:
- Open, Soft, Filter with Excel
- Import into SPLUNK
- SIMILE
- Tapestry

log2timeline PARSING PLUGINS

apache2_error - Apache2 error log file
chrome - Chrome history file
encase_dirlisting - CSV file that is exported from encase
evt - Windows 2k/XP/2k3 Event Log
evtx - Windows Event Log File (EVTX)
exif - Metadata information from files using ExifTool
ff_bookmark - Firefox bookmark file
firefox2 - Firefox 2 browser history
firefox3 - Firefox 3 history file
ftk_dirlisting - CSV file that is exported from FTK Imager (dirlisting)
generic_linux - Generic Linux logs that start with MMM DD HH:MM:SS
iehistory - index.dat file containg IE history
iis - IIS W3C log file
isatxt - ISA text export log file
jp_ntfs_change - CSV output file from JP (NTFS Change log)
mactime - Body file in the mactime format
mcafee - Log file
mft - NTFS MFT file
mssql_errlog - ERRORLOG file produced by MS SQL server
ntuser - NTUSER.DAT registry file
opera - Opera's global history file
oxml - OpenXML document pcap
pcap - PCAP file
pdf - Available PDF document metadata
prefetch - Prefetch directory
recycler - Recycle bin directory
restore 0.9 - Restore point directory
safari - Safari History.plist file
sam - SAM registry file
security - SECURITY registry file
setupapi - SetupAPI log file in Windows XP
skype_sql - Skype database
software - SOFTWARE registry file
sol - .sol (LSO) or a Flash cookie file
squid - Squid access log (http_emulate off)
syslog - Linux Syslog log file
system - SYSTEM registry file
tln - Body file in the TLN format
volatility - Volatility output files (psscan2, sockscan2, ...)
win_link - Windows shortcut file (or a link file)
wmiprov - wmiprov log file
xpfirewall - XP Firewall log

List plugins # log2timeline -f list
...HELP EXPAND THIS LIST. BUILD PLUGINS!!!

BY DAVID NIDES (12/16/2011)
TWITTER: @DAVNADS
BLOG: DAVNADS.BLOGSPOT.COM
EMAIL: DNIDES@KPMG.COM

CREDITS TO: ED GOINGS, ROB LE... KRISTINN GUDJONSSON, KPMG &...!!
QUESTIONS/FEEDBACK–CONTACT US!

KEY

Red text – image/source
Blue text – mount point
Purple text – output file
Green text – log2timeline plugins
Brown text - TimeZone

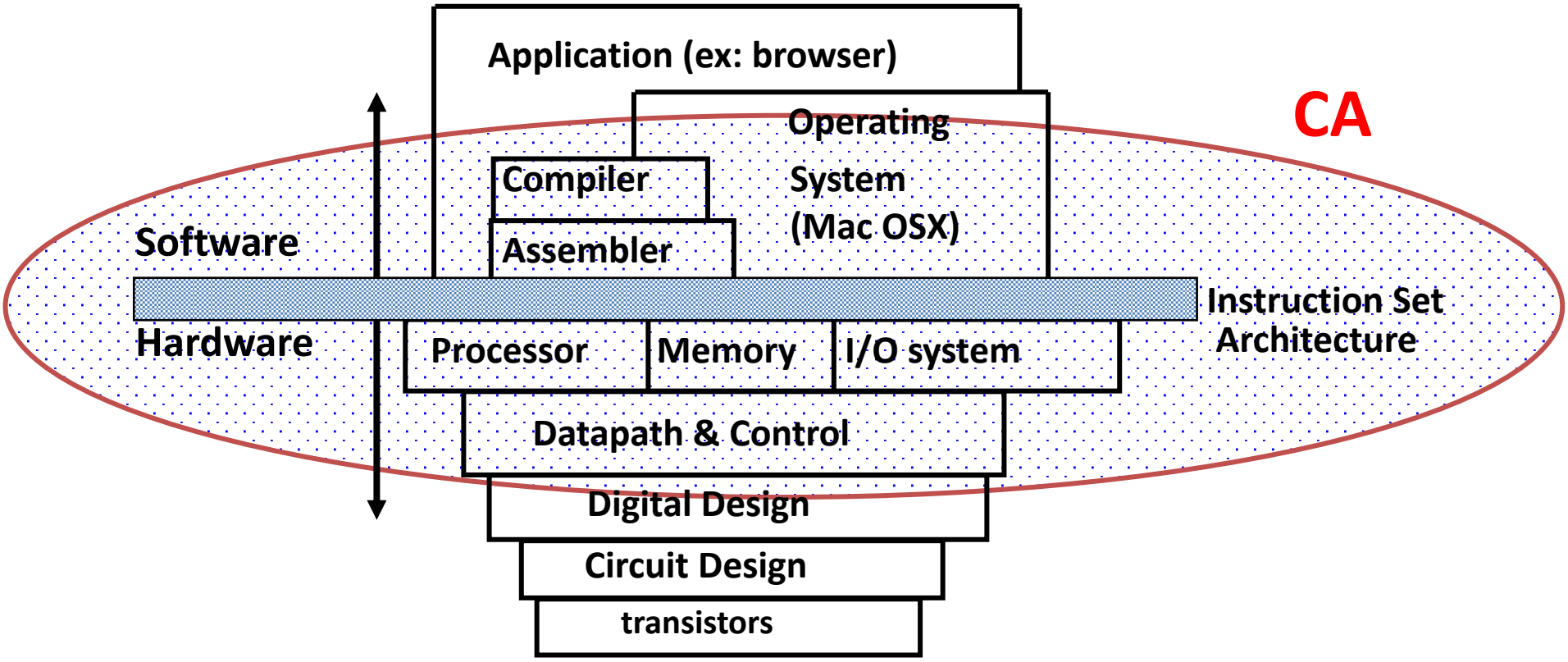
Content

- Main topics: Everything till (including) Lecture 16
  - Number representation (int & float (Lecture 17!))
  - C
  - CALL
  - RISC-V
  - SDS; Datapath & Control
  - Pipelining & Superscalar
  - Caches
- Plus general ”Computer Architecture” knowledge
- Disclaimer: In this review, important topics for CA are covered. It does not indicate that other topics from lectures 1-16 will not covered in the exams, nor does it mean that everything written here will be covered.

New School Computer Architecture (1/3)

Personal Mobile Devices

Old Machine Structures



New-School Machine Structures (It's a bit more complicated!)

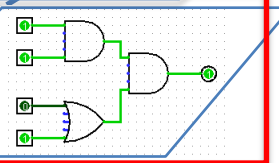
Software | *Hardware*

- Parallel Requests
  - Assigned to computer
  - e.g., Search "Katz"
- Parallel Threads
  - Assigned to core
  - e.g., Lookup, Ads
- Parallel Instructions
  - >1 instruction @ one time
  - e.g., 5 pipelined instructions
- Parallel Data
  - >1 data item @ one time
  - e.g., Add of 4 pairs of words
- Hardware descriptions
  - All gates functioning in parallel at same time
- Programming Languages

Leverage Parallelism & Achieve High Performance

Warehouse Scale Computer

Smart Phone



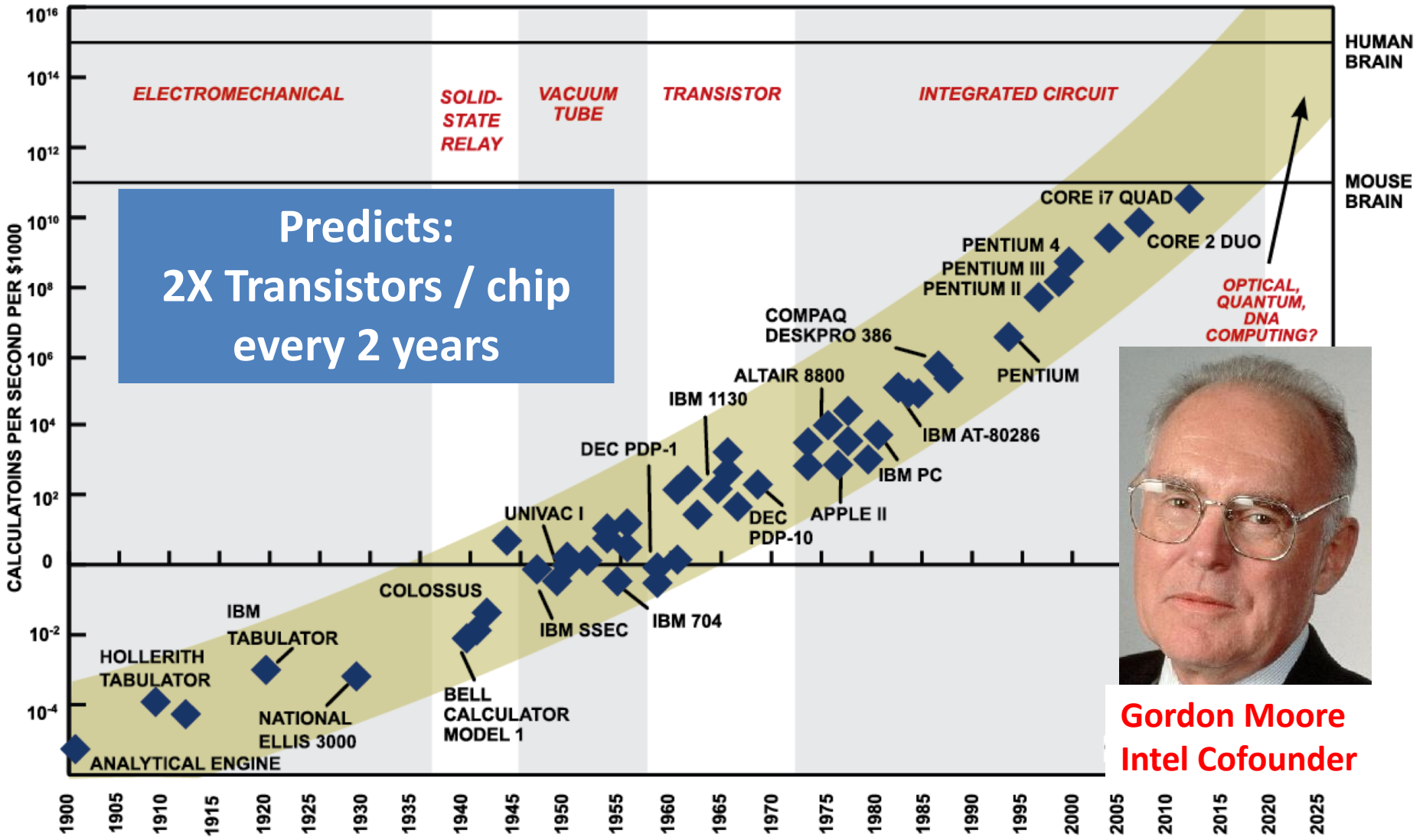
CA is NOT about C Programming

- It's about the hardware-software interface
  - What does the programmer need to know to achieve the highest possible performance
- Languages like C are closer to the underlying hardware, unlike languages like Python!
  - Allows us to talk about key hardware features in higher level terms
  - Allows programmer to explicitly harness underlying hardware parallelism for high performance: "programming for performance"

Great Ideas in Computer Architecture

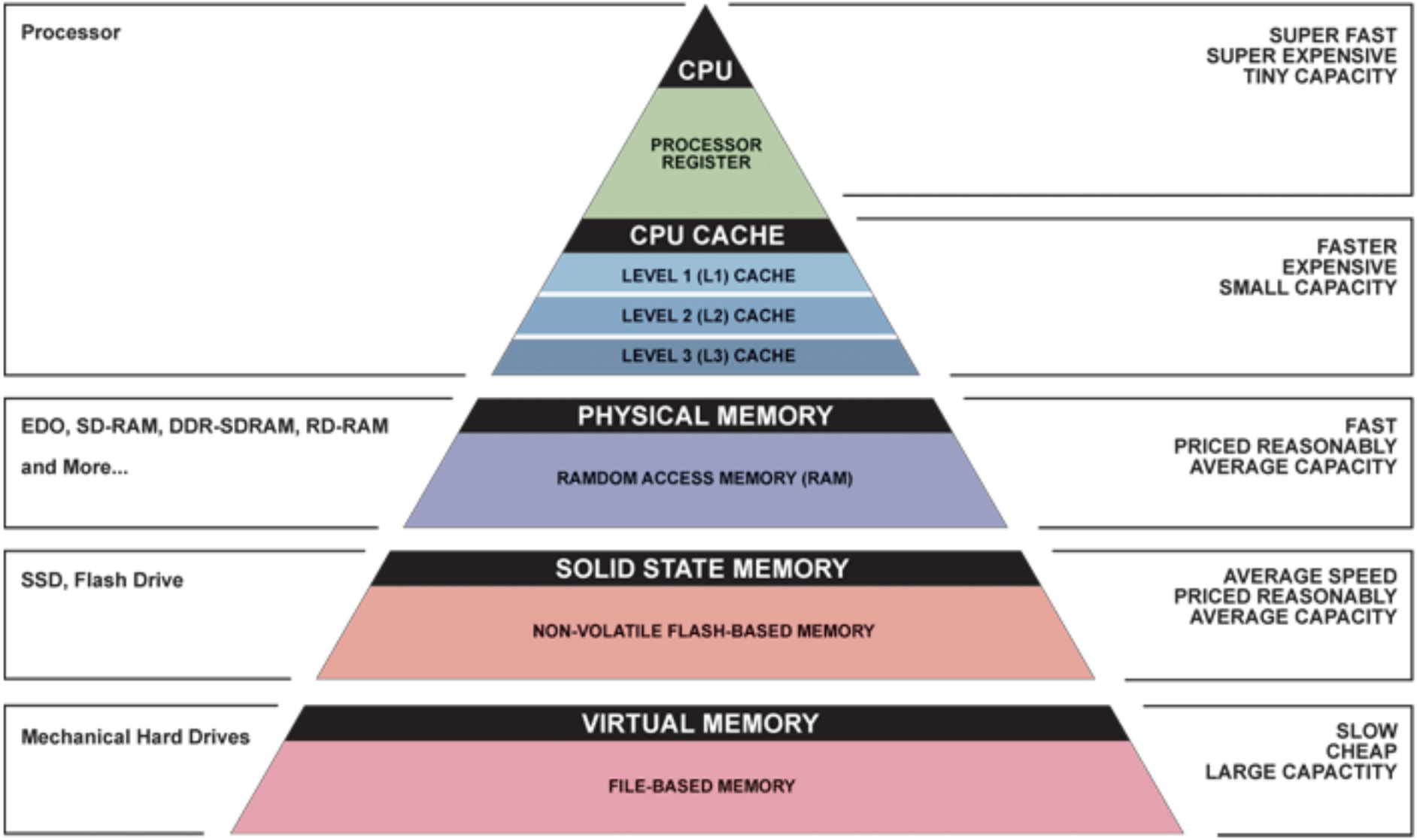
1. *Design for Moore's Law*
   -- Higher capacities caches and DRAM
2. Abstraction to Simplify Design
3. Make the Common Case Fast
4. *Dependability via Redundancy*
   -- Parity, SEC/DEC
5. *Memory Hierarchy*
   -- Caches, TLBs
6. *Performance via Parallelism/Pipelining/Prediction*
   -- Data-level Parallelism

#2: Moore's Law

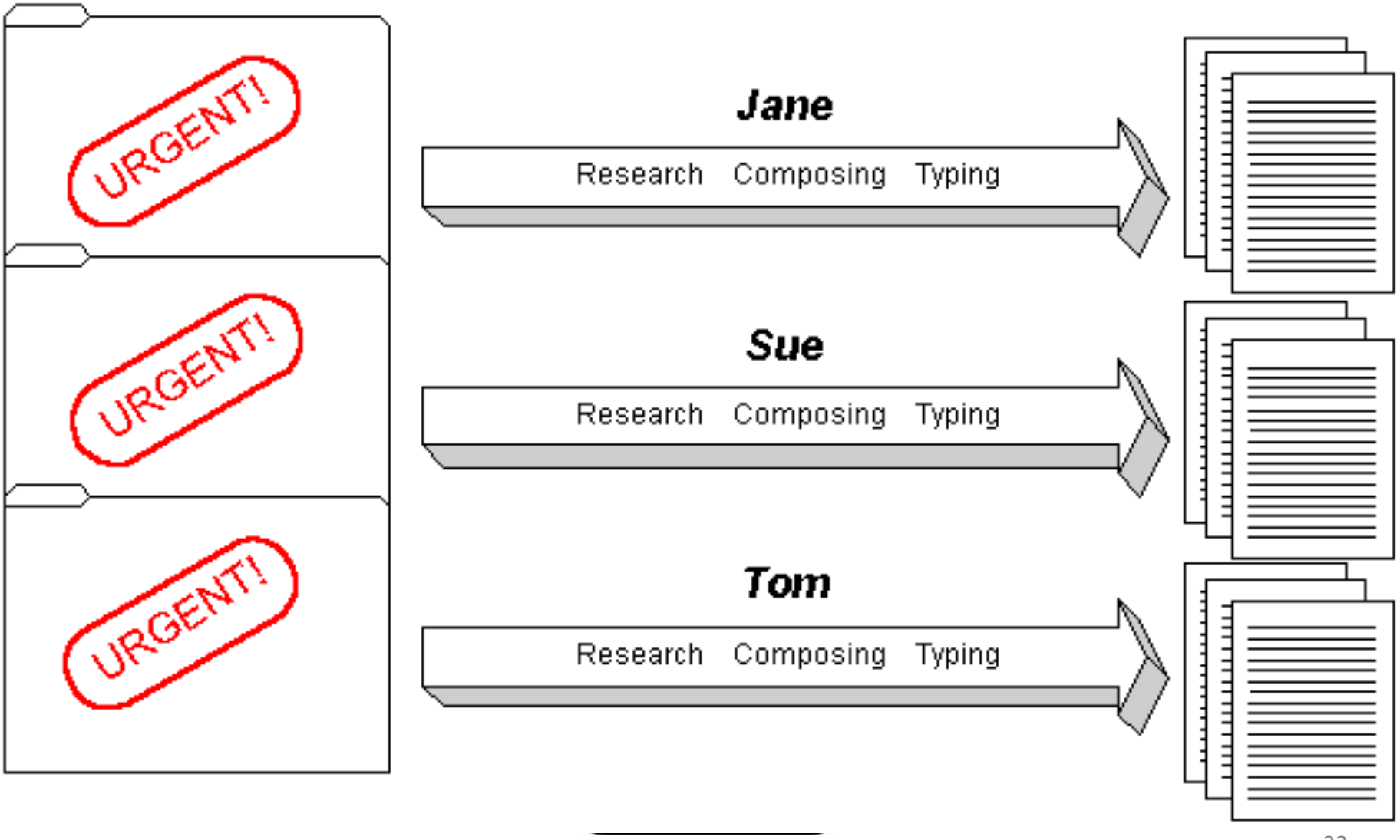


SOURCE: RAY KURZWEIL, "THE SINGULARITY IS NEAR: WHEN HUMANS TRANSCEND BIOLOGY", P.67, *THE VIKING PRESS*, 2006. DATAPOINTS BETWEEN 2000 AND 2012 REPRESENT BCA ESTIMATES.

Great Idea #3: Principle of Locality/ Memory Hierarchy



Great Idea #4: Parallelism

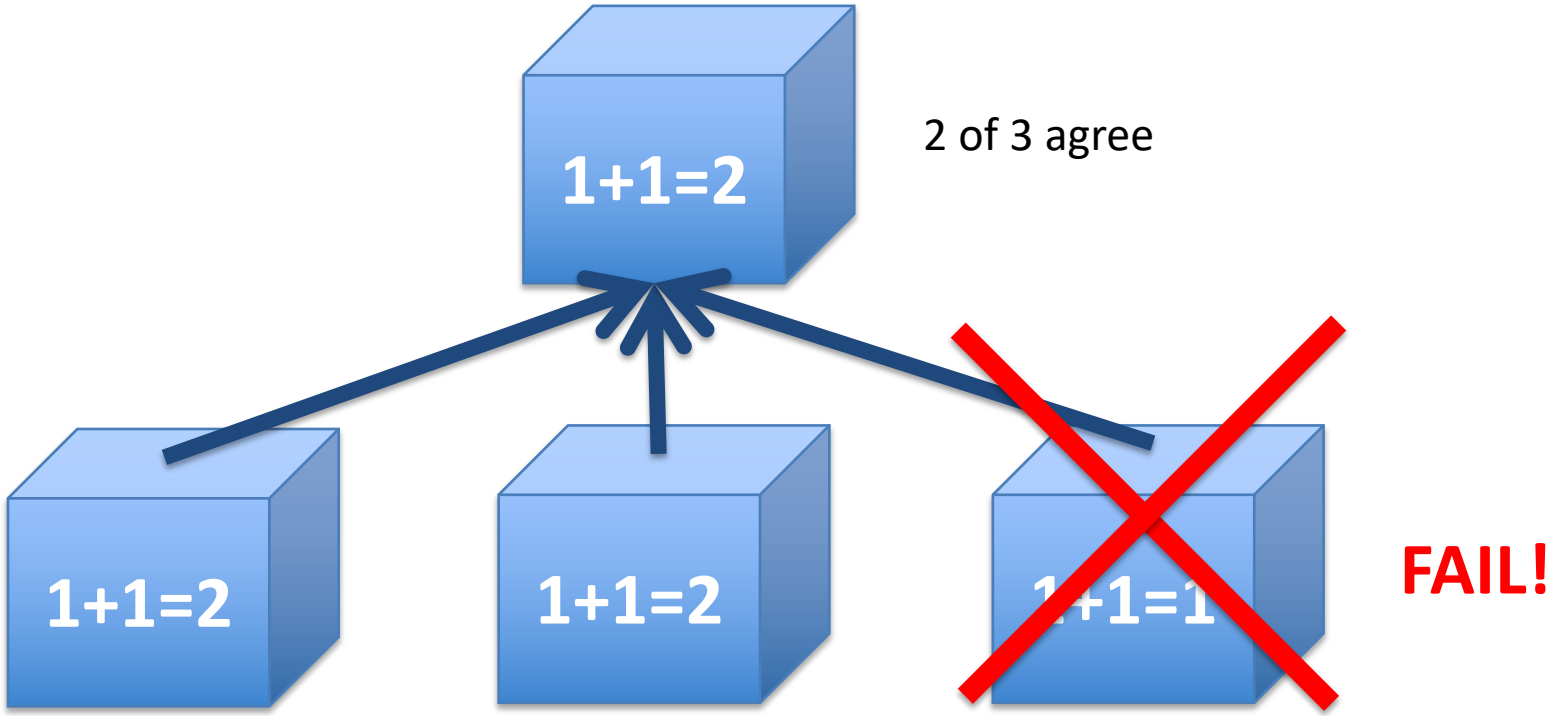


Great Idea #5: Performance Measurement and Improvement

- Tuning application to underlying hardware to exploit:
  - Locality
  - Parallelism
  - Special hardware features, like specialized instructions (e.g., matrix manipulation)
- Latency
  - How long to set the problem up
  - How much faster does it execute once it gets going
  - It is all about *time to finish*

Great Idea #6: Dependability via Redundancy

- Redundancy so that a failing piece doesn't make the whole system fail



Increasing transistor density reduces the cost of redundancy

Key Concepts

- Inside computers, everything is a number
- But numbers usually stored with a fixed size
  - 8-bit bytes, 16-bit half words, 32-bit words, 64-bit double words, …
- Integer and floating-point operations can lead to results too big/small to store within their representations: *overflow*/*underflow*

Number Representation

Signed Integers and Two's-Complement Representation

- Signed integers in C; want ½ numbers <0, want ½ numbers >0, and want one 0
- *Two's complement* treats 0 as positive, so 32-bit word represents 2^{32} integers from -2^{31} (–2,147,483,648) to $2^{31}-1$ (2,147,483,647)
  - Note: one negative number with no positive version
  - Book lists some other options, all of which are worse
  - Every computer uses two's complement today
- *Most-significant bit* (leftmost) is the *sign bit*, since 0 means positive (including 0), 1 means negative
  - Bit 31 is most significant, bit 0 is least significant

Two's-Complement Integers

Sign Bit

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = 1_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = 2_{ten}$
… …
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = 2{,}147{,}483{,}645_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0010_{two} = -2{,}147{,}483{,}646_{ten}$
… …
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1101_{two} = -3_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -1_{ten}$

Ways to Make Two's Complement

- For N-bit word, complement to 2_{ten}^{N}
  - For 4 bit number $3_{ten}=0011_{two}$, two's complement

    (i.e. -3_{ten}) would be

    $16_{ten}-3_{ten}=13_{ten}$ or $10000_{two} - 0011_{two} = 1101_{two}$
- Here is an easier way:
  - Invert all bits and add 1
  - Computers actually do it like this, too

3_{ten} 0011_{two}

Bitwise complement 1100_{two}

$+ \quad 1_{two}$

-3_{ten} 1101_{two}

Two's-Complement Examples

- Assume for simplicity 4 bit width, -8 to +7 represented

```
 3   0011        3   0011        -3   1101
+2   0010   + (-2) 1110   + (-2) 1110
 5   0101      1 1 0001      -5 1 1011
```

Overflow when magnitude of result too big small to fit into result representation

```
 7   0111        -8   1000
+1   0001   + (-1) 1111
-8   1000      +7 1 0111
```

Overflow! ***Overflow!***

Carry into MSB = Carry Out MSB

Carry into MSB ≠ Carry Out MSB

Carry in = carry from less significant bits
Carry out = carry to more significant bits

Suppose we had a 5-bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ **-15 to +16**

Suppose we had a 5-bit word. What integers can be represented in two's complement?

- ☐ -32 to +31
- ☐ 0 to +31
- ☐ -16 to +15
- ☐ -15 to +16

www.h-schmidt.net/FloatApplet/IEEE754.html

Conclusion

Exponent tells Significand how much (2^i) to count by (…, 1/4, 1/2, 1, 2, …)

- Floating Point lets us:
  - Represent numbers containing both integer and fractional parts; makes efficient use of available bits.
  - Store approximate values for very large and very small #s.
- IEEE 754 Floating-Point Standard is most widely accepted attempt to standardize interpretation of such numbers (Every desktop or server computer sold since ~1997 follows these conventions)

Can store NaN, ± ∞

- **Summary (single precision):**

- **$(-1)^S \times (1 + \text{Significand}) \times 2^{(\text{Exponent}-127)}$**
- **Double precision identical, except with exponent bias of 1023 (half, quad similar)**

Float: Special Numbers Summary

- **Reserve exponents, significands:**

Exponent	Significand	Object
0	0	0
0	nonzero	Denorm
1-254	anything	+/- fl. pt. #
255	0	+/- ∞
255	nonzero	NaN

C Programming

Quiz: Pointers

```
void foo(int *x, int *y)
{  int t;
   if ( *x > *y ) { t = *y; *y = *x; *x = t; }
}
int a=3, b=2, c=1;
foo(&a, &b);
foo(&b, &c);
foo(&a, &b);
printf("a=%d b=%d c=%d\n", a, b, c);
```

Result is:

A: **a=3 b=2 c=1**

B: **a=1 b=2 c=3**

C: **a=1 b=3 c=2**

D: **a=3 b=3 c=3**

E: **a=1 b=1 c=1**

Arrays and Pointers

```
int
foo(int array[],
    unsigned int size)
{
   …
   printf("%d\n", sizeof(array));
}

int
main(void)
{
   int a[10], b[5];
   int c[] = {1, 3, 2, 5, 6};
   … foo(a, 10)… foo(c, 5) …
   printf("%d\n", sizeof(c));
}
```

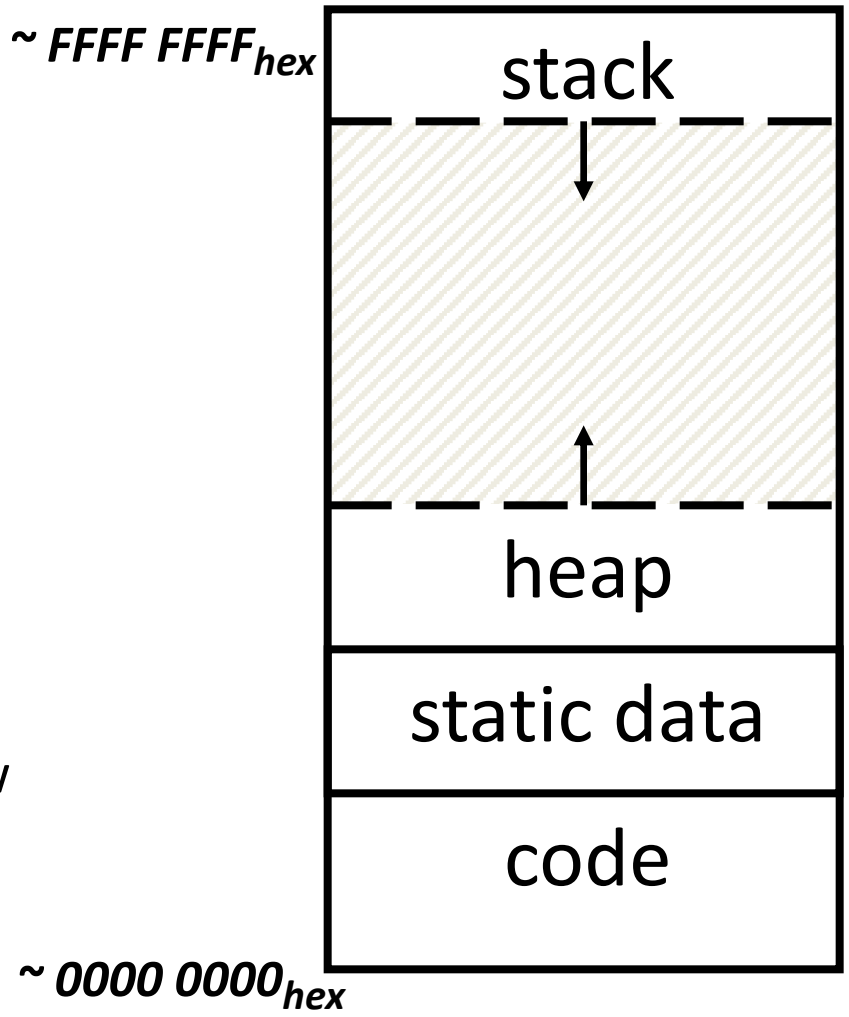
What does this print (64bit) ? **8**

... because **`array`** is really a pointer (and a pointer is architecture dependent, but 8 on 64bit machines!)

What does this print? **40**

C Memory Management

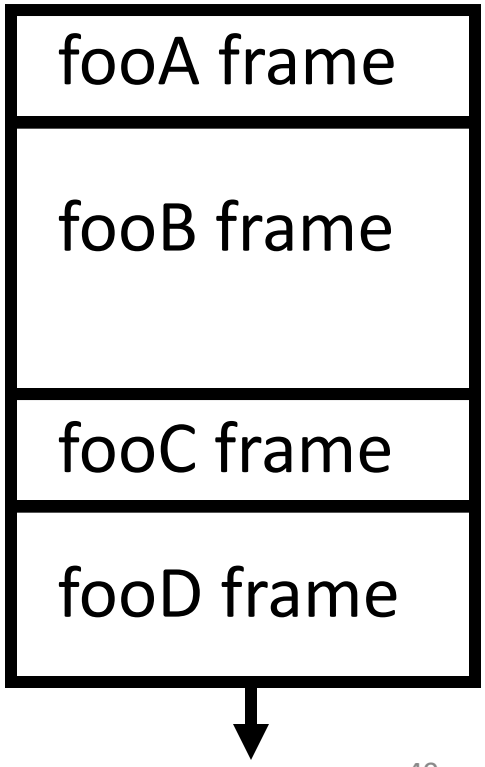
- Program's *address space* contains 4 regions:
  - stack: local variables inside functions, grows downward
  - heap: space requested for dynamic data via **`malloc()`**; resizes dynamically, grows upward
  - static data: variables declared outside functions, does not grow or shrink. Loaded when program starts, can be modified.
  - code: loaded when program starts, does not change



The Stack

- Every time a function is called, a new frame is allocated on the stack
- Stack frame includes:
  - Return address (who called me?)
  - Arguments
  - Space for local variables
- Stack frames contiguous blocks of memory; stack pointer indicates start of stack frame
- When function ends, stack frame is tossed off the stack; frees memory for future stack frames
- We'll cover details later for RISC-V processor

```
fooA() { fooB(); }
fooB() { fooC(); }
fooC() { fooD(); }
```



Faulty Heap Management

- What is wrong with this code?
- Memory leak!

```
int foo() {
   int *value = malloc(sizeof(int));
   *value = 42;
   return *value;
}
```

And In Conclusion, …

- Pointers are an abstraction of machine memory addresses
- Pointer variables are held in memory, and pointer values are just numbers that can be manipulated by software
- In C, close relationship between array names and pointers
- Pointers know the type of the object they point to (except void *)
- Pointers are powerful but potentially dangerous

RISC-V

Addition and Subtraction of Integers (3/4)

- How to do the following C statement?

  a = b + c + d - e; // a: x10; b: x1; c: x2, e: x3; f: x4

- Break into multiple instructions

```
add x10, x1, x2   # a_temp = b + c
add x10, x10, x3  # a_temp = a_temp + d
sub x10, x10, x4  # a = a_temp - e
```

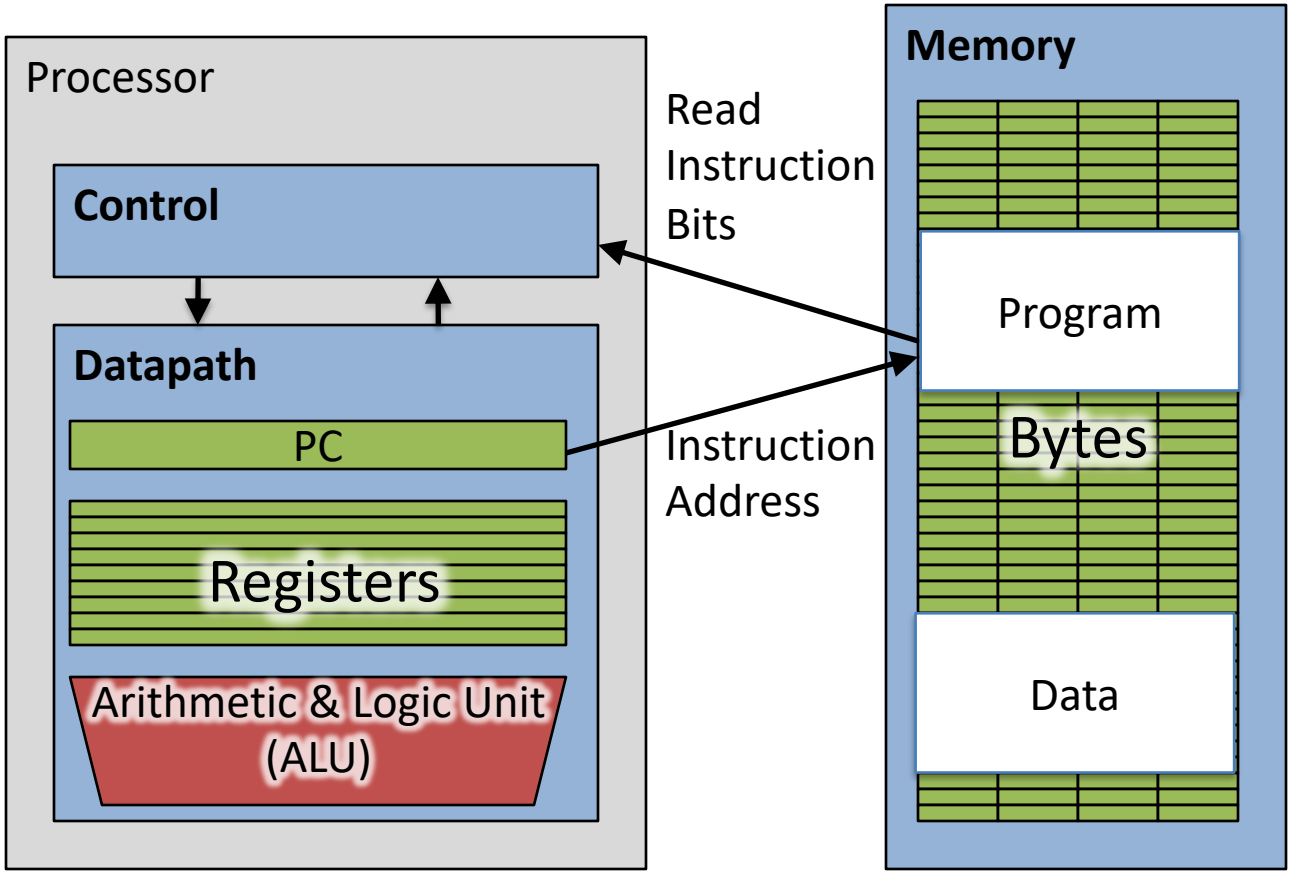
- Notice: A single line of C may break up into several lines of RISC-V.
- Notice: Everything after the hash mark on each line is ignored (comments).

Question:

We want to translate `*x = *y +1` into RISC-V
(`x`, `y` int pointers stored in: `s0 s1`)

```
A:    addi s0, s1, 1

B:    lw   s0, 1(s1)
      sw   s1, 0(s0)

C:    lw   t0, 0(s1)
      addi t0, t0, 1
      sw   t0, 0(s0)


D:    sw   t0, 0(s1)
      addi t0, t0, 1
      lw   t0, 0(s0)

E:    lw   s0, 1(t0)
      sw   s1, 0(t0)
```

Executing a Program



- The PC (program counter) is internal register inside processor holding byte address of next instruction to be executed.
- Instruction is fetched from memory, then control unit executes instruction using datapath and memory system, and updates program counter (default is add +4 bytes to PC, to move to next sequential instruction)

Question!

```
        addi s0,zero,0
Start:  slt  t0,s0,s1
        beq  t0,zero,Exit
        sll  t1,s0,2
        add  t1,t1,s5
        lw   t1,0(t1)
        add  s4,s4,t1
        addi s0,s0,1
        j Start
Exit:
```

What is the code above?

A: while loop

B: do … while loop

C: for loop

D: A or C

E: Not a loop

Question

- What value does x12 have at the end?
- Answer:
  x12 = 16

```c
#include <stdio.h>

int main (){
    int x10 = 7;
    int x12 = 0;
    do{
        int x14 = x10 & 1;
        if(x14)
            x12 += x10;

        x10--;
    } while (x10 != 0);
    printf("%d", x12);
}
```

```
    addi   x10, x0 ,  0x07
    add    x12, x0 ,  x0
label_a:
    andi   x14, x10, 1
    beq    x14, x0 , label_b
    add    x12, x10, x12
label_b:
    addi   x10, x10, -1
    bne    x10, x0 , label_a
```

RISC-V Function Call Conventions

- Registers faster than memory, so use them
- Give names to registers, conventions on how to use them
- `a0–a7 (x10–x17)`: eight *argument* registers to pass parameters and return values `(a0–a1)`
- `ra`: one *return address* register to return to the point of origin (`x1`)
- Also `s0–s1 (x8–x9)` and `s2–s11 (x18–x27)`: saved registers (more about those later)

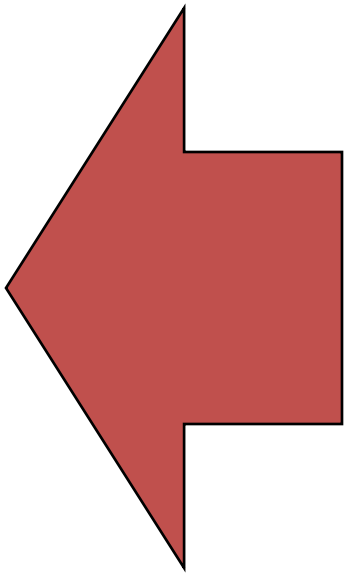
Instruction Support for Functions (1/4)

C

```
... sum(a,b);... /* a, b: s0, s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

RISC-V

```
address (shown in decimal)
 1000
 1004
 1008
 1012
 1016
 …
 2000
 2004
```



In RV32, instructions are 4 bytes, and stored in memory just like data. So here we show the addresses of where the programs are stored.

Instruction Support for Functions (2/4)

C

```
... sum(a,b);... /* a, b: s0, s1 */
 }
 int sum(int x, int y) {
   return x+y;
 }
```

RISC-V

```
address (shown in decimal)
 1000 add  a0, s0, x0       # x = a
 1004 mv   a1, s1           # y = b
 1008 addi ra, zero, 1016   # ra=1016
 1012 j    sum              # jump to sum
 1016 …                     # next instruction
 …
 2000 sum: add a0, a0, a1
 2004 jr   ra     # new instr. "jump register"
```

Instruction Support for Functions (3/4)

C

```
... sum(a,b);... /* a,b:$s0,$s1 */
}
int sum(int x, int y) {
  return x+y;
}
```

- Question: Why use `jr` here? Why not use `j`?
- Answer: `sum` might be called by many places, so we can't return to a fixed place. The calling proc to `sum` must be able to say "return here" somehow.

RISC-V

```
2000 sum: add a0, a0, a1
2004 jr   ra  # new instr. "jump register"
```

Instruction Support for Functions (4/4)

- Single instruction to jump and save return address: jump and link (**`jal`**)
- Before:

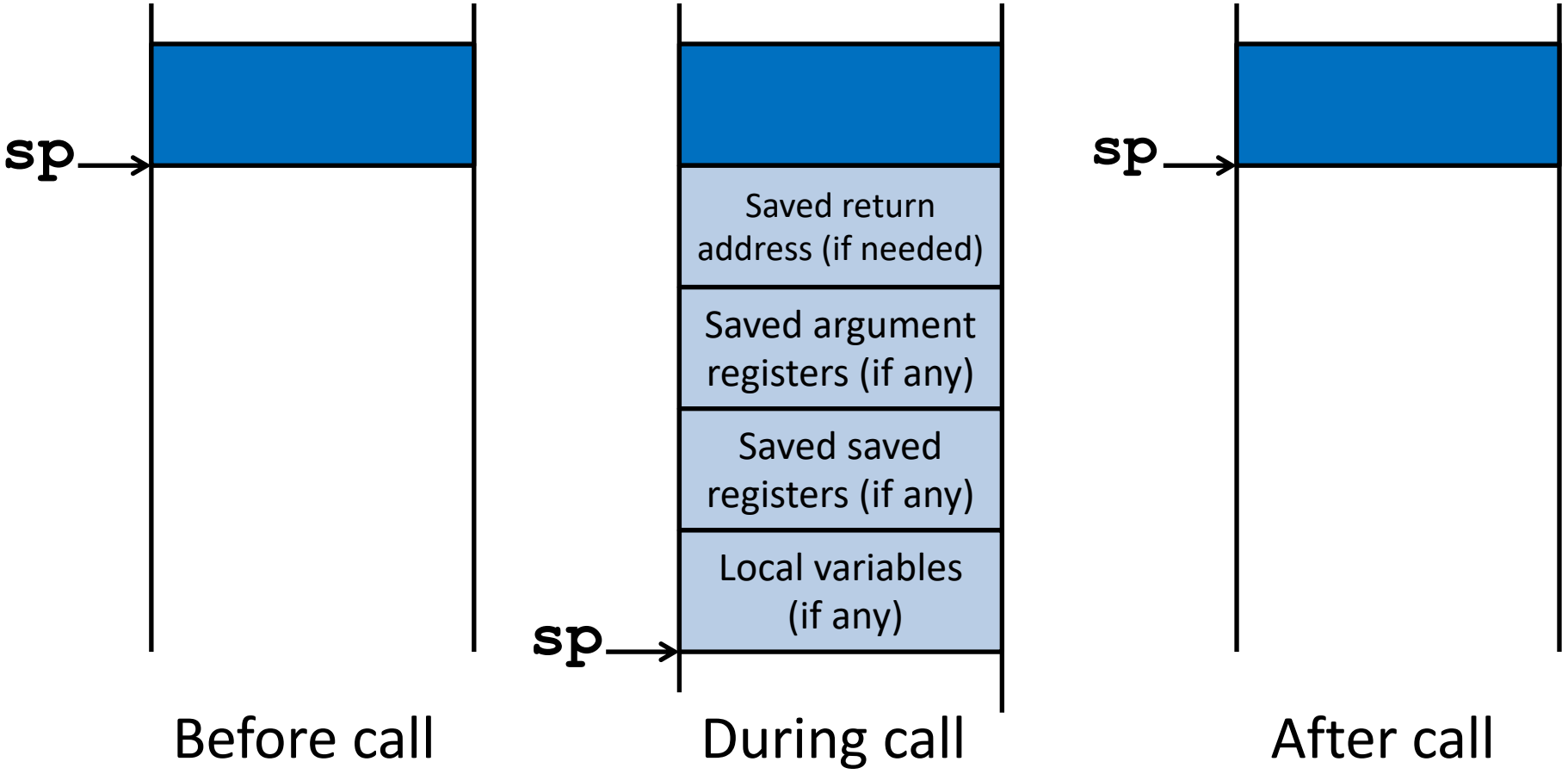
```
1008 addi ra, zero, 1016   # $ra=1016
1012 j sum                 # goto sum
```

- After:

```
1008 jal sum   # ra=1012, goto sum
```

- Why have a **`jal`**?
  - Make the common case fast: function calls very common.
  - Reduce program size
  - Don't have to know where code is in memory with **`jal`**!

Stack Before, During, After Function



Using the Stack (1/2)

- We have a register **`sp`** which always points to the last used space in the stack.
- To use stack, we decrement this pointer by the amount of space we need and then fill it with info.
- So, how do we compile this?

```
int sumSquare(int x, int y) {
   return mult(x,x)+ y;
}
```

Using the Stack (2/2)

```
int sumSquare(int x, int y) {
  return mult(x,x)+ y; }
```

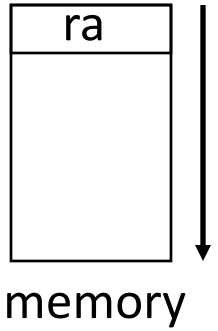
```
  sumSquare:
"push"  addi   sp, sp, -8   # space on stack
        sw     ra, 4(sp)    # save ret addr
        sw     a1, 0(sp)    # save y
        mv     a1, a0       # mult(x,x)
        jal    mult         # call mult
        lw     a1, 0(sp)    # restore y
"pop"   add    a0, a0, a1   # mult()+y
        lw     ra, 4(sp)    # get ret addr
        addi   sp, sp, 8    # restore stack
        jr ra
  mult: ...
```

Basic Structure of a Function

Prologue

```
entry_label:
addi sp,sp, -framesize
sw   ra, framesize-4(sp)   # save ra
save other regs if need be
```

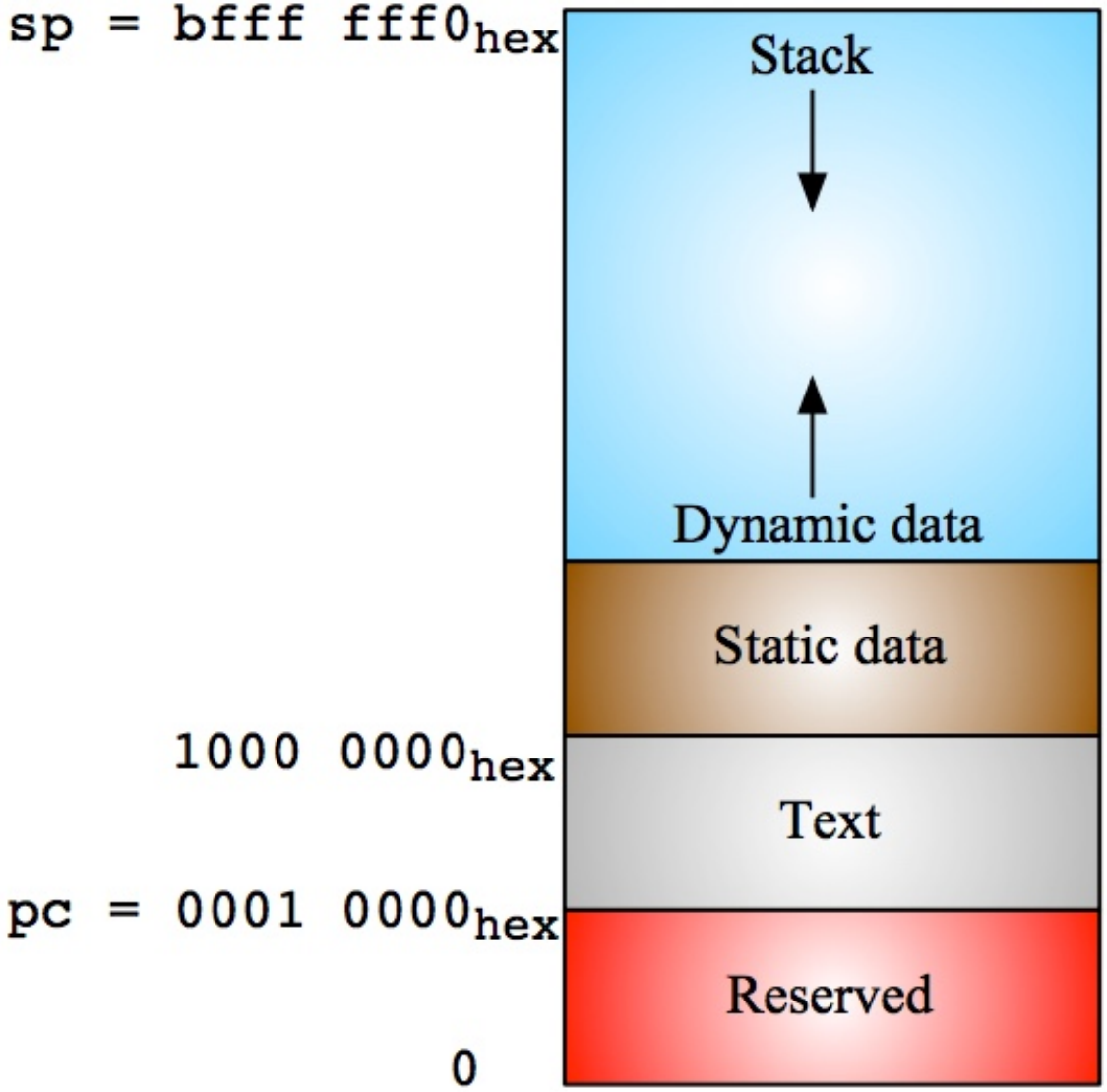
Body `...` **(call other functions…)**



Epilogue

```
restore other regs if need be
lw   ra, framesize-4(sp)   # restore $ra
addi sp, sp, framesize
jr ra
```

RV32 Memory Allocation



RISC-V ISA so far…

- Registers we know so far (All of them!)
  - a0-a7 for function arguments, a0-a1 for return values
  - sp, stack pointer, ra return address
  - s0-s11 saved registers
  - t0-t6 temporaries
  - zero
- Instructions we know:
  - Arithmetic: add, addi, sub
  - Logical: sll, srl, slli, srli, slai, and, or, xor, andi, ori, xori
  - Decision: beq, bne, blt, bge
  - Unconditional branches (jumps): j, jr
  - Functions called with `jal`, return with `jr ra`.
- The stack is your friend: Use it to save anything you need. Just leave it the way you found it!

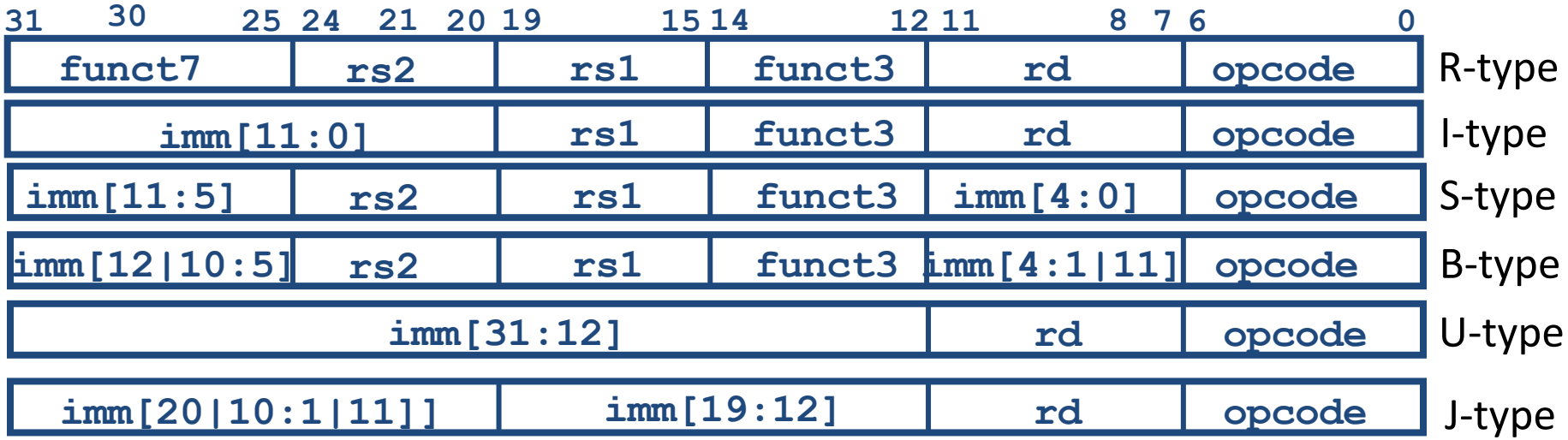
12 Shift Instructions…

- Two versions of of all shift instructions. Shift amount via:
  - Register
  - Immediate
- (On RV64: additional "word" version of instruction: only works on first 32bit of 64bit register)
- Shift Left
- Shift Right Arithmetic: Fill upper bits with **msb**
- Shift Right Logic: Fill upper bits with 0's

`sll,sllw`	R	Shift Left (Word)	R[rd] = R[rs1] << R[rs2]	1)
`slli,slliw`	I	Shift Left Immediate (Word)	R[rd] = R[rs1] << imm	1)
`sra,sraw`	R	Shift Right Arithmetic (Word)	R[rd] = R[rs1] >> R[rs2]	1,5)
`srai,sraiw`	I	Shift Right Arith Imm (Word)	R[rd] = R[rs1] >> imm	1,5)
`srl,srlw`	R	Shift Right (Word)	R[rd] = R[rs1] >> R[rs2]	1)
`srli,srliw`	I	Shift Right Immediate (Word)	R[rd] = R[rs1] >> imm	1)

Notes: 1) The Word version only operates on the rightmost 32 bits of a 64-bit registers
5) Replicates the sign bit to fill in the leftmost bits of the result during right shift

Summary of RISC-V Instruction Formats



CORE INSTRUCTION FORMATS

	31 27	26 25	24 20	19 15	14 12	11 7	6 0
R	funct7		rs2	rs1	funct3	rd	Opcode
I	imm[11:0]			rs1	funct3	rd	Opcode
S	imm[11:5]		rs2	rs1	funct3	imm[4:0]	opcode
SB	imm[12|10:5]		rs2	rs1	funct3	imm[4:1|11]	opcode
U	imm[31:12]					rd	opcode
UJ	imm[20|10:1|11|19:12]					rd	opcode

Question

- What is correct encoding of add x4, x3, x2 ?

  A: $4021\ 8233_{hex}$

  B: $0021\ 82b3_{hex}$

  C: $4021\ 82b3_{hex}$

  D: $0021\ 8233_{hex}$

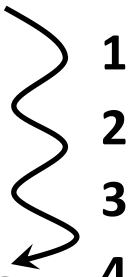
  E: $0021\ 8234_{hex}$

31 25	24 20	19 15	14 12	11 7	6 0	
0000000	rs2	rs1	000	rd	0110011	add
0100000	rs2	rs1	000	rd	0110011	sub
0000000	rs2	rs1	100	rd	0110011	xor
0000000	rs2	rs1	110	rd	0110011	or
0000000	rs2	rs1	111	rd	0110011	and

Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End:  # target instruction
```



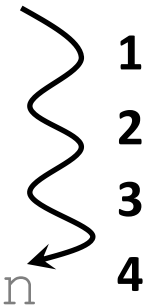
Count instructions from branch

- Branch offset = **4×32-bit instructions = 16 bytes**
- (Branch with offset of 0, branches to itself)

Branch Example, Determine Offset

- RISC-V Code:

```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End:  # target instruction
```



Count instructions from branch

???????	01010	10011	000	?????	1100011
imm	**rs2=10**	**rs1=19**	**BEQ**	**imm**	**BRANCH**

Branch Example, Encode Offset

- RISC-V Code:

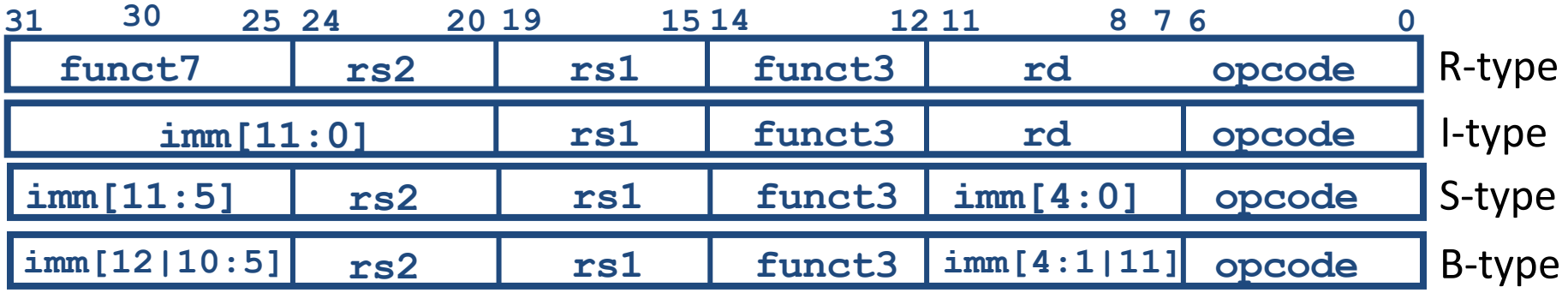
```
Loop: beq   x19,x10,End
      add   x18,x18,x10
      addi  x19,x19,-1
      j     Loop
End:  # target instruction
```

offset = 16 bytes = 8x2 bytes

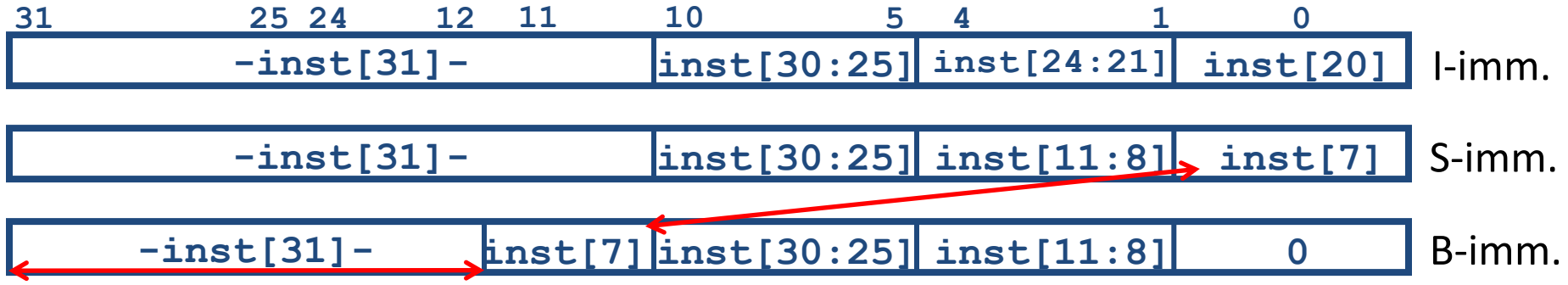
???????	01010	10011	000	?????	1100011
imm	**rs2=10**	**rs1=19**	**BEQ**	**imm**	**BRANCH**

RISC-V Immediate Encoding

Instruction encodings, inst[31:0]



32-bit immediates produced, imm[31:0]

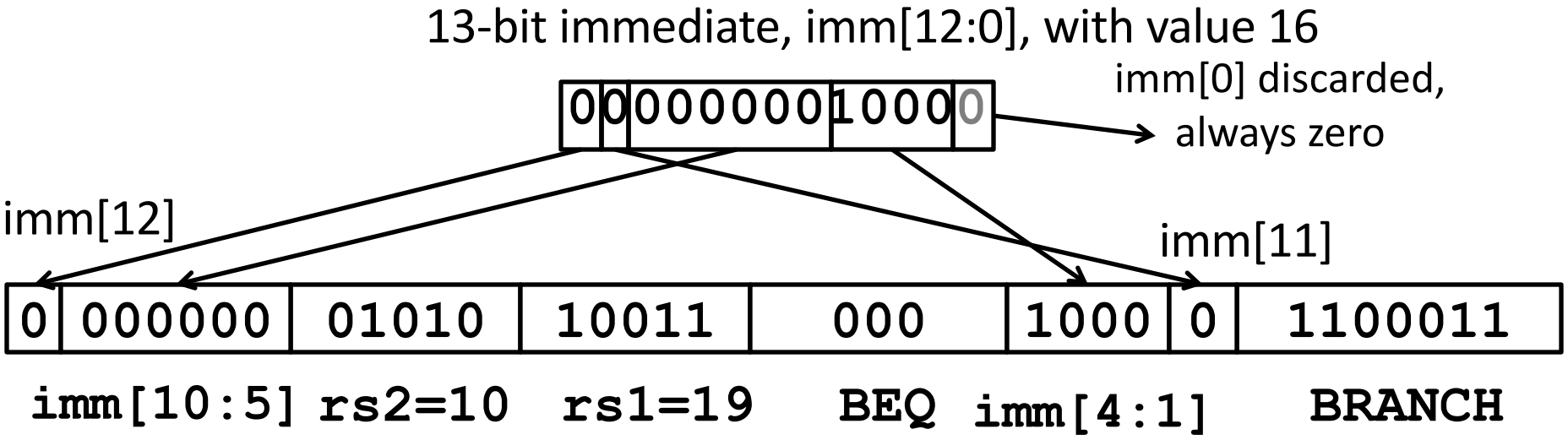


Upper bits sign-extended from inst[31] always

Only bit 7 of instruction changes role in immediate between S and B

Branch Example, complete encoding

beq x19,x10, offset = 16 bytes



LUI to Create Long Immediates

- LUI writes the upper 20 bits of the destination with the immediate value, and clears the lower 12 bits.
- Together with an ADDI to set low 12 bits, can create any 32-bit value in a register using two instructions (LUI/ADDI).

```
LUI x10, 0x87654     # x10 = 0x87654000
ADDI x10, x10, 0x321 # x10 = 0x87654321
```

One Corner Case

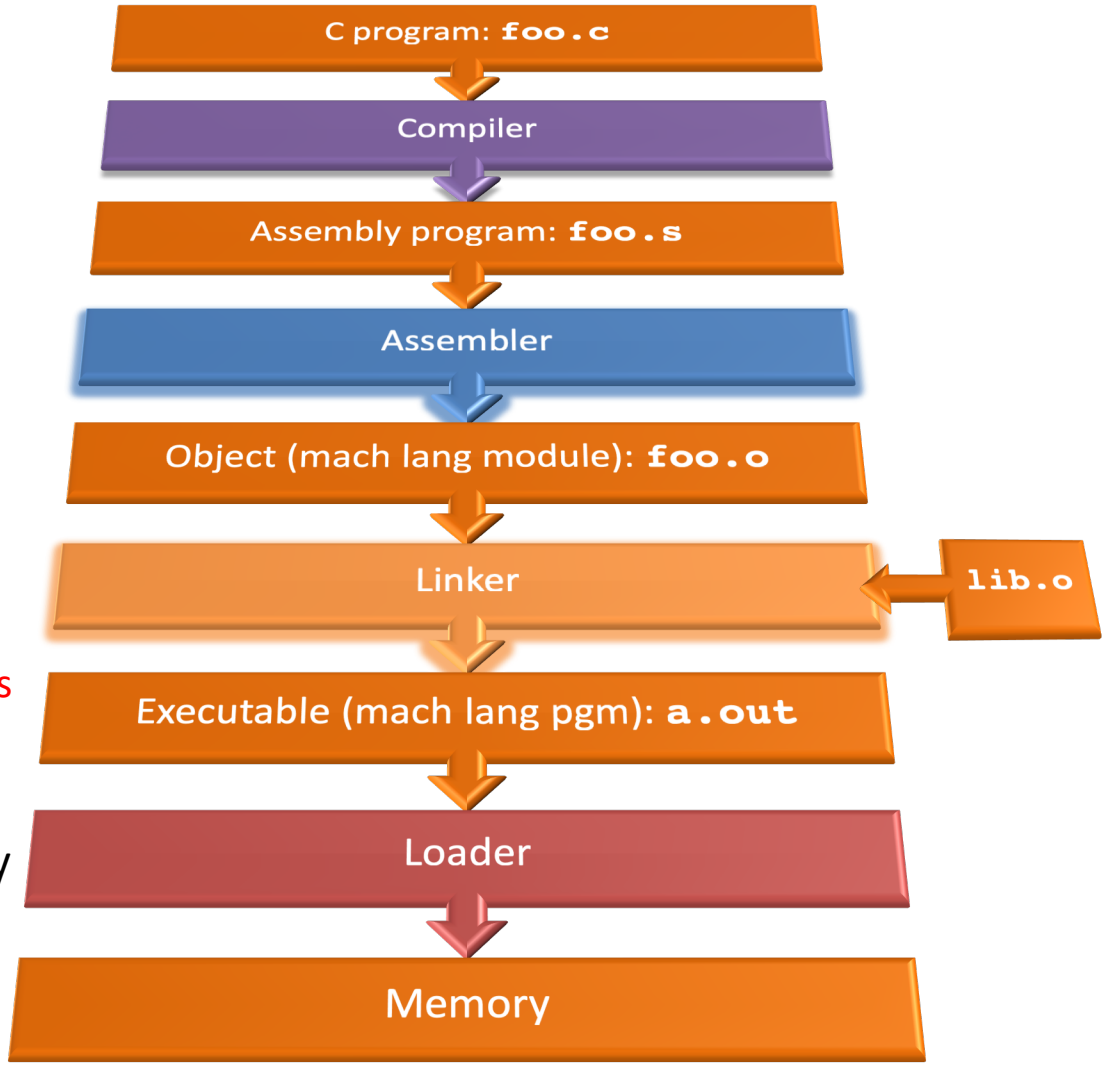
How to set 0xDEADBEEF?

```
LUI x10, 0xDEADB      # x10 = 0xDEADB000
ADDI x10, x10, 0xEEF  # x10 = 0xDEADAEEF
```

ADDI 12-bit immediate is always sign-extended, if top bit is set, will subtract 1 from upper 20 bits

Steps in compiling a C program

- Compiler converts a single HLL file into a single assembly language file.
- Assembler removes pseudo-instructions, converts what it can to machine language, and creates a checklist for the linker (relocation table). A `.s` file becomes a `.o` file.
  - Does 2 passes to resolve addresses, handling internal forward references
- Linker combines several `.o` files and resolves absolute addresses.
  - Enables separate compilation, libraries that need not be compiled, and resolves remaining addresses
- Loader loads executable into memory and begins execution.



Question

At what point in process are all the machine code bits generated for the following assembly instructions:

1) `add 6, 7, 8`
2) `jal fprintf`

A: 1) & 2) After compilation

B: 1) After compilation, 2) After assembly

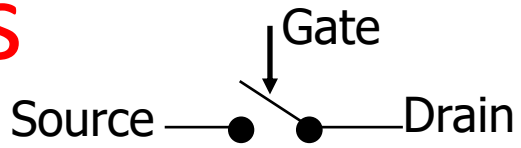
C: 1) After assembly, 2) After linking

D: 1) After assembly, 2) After loading

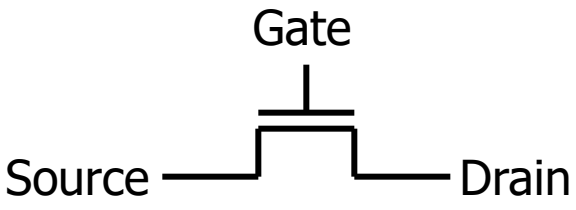
E: 1) After compilation, 2) After linking

SDS/ RISC-V Pipline

CMOS Transistors



- Three terminals: source, gate, and drain
  - Switch action:
    if voltage on gate terminal is (some amount) higher/lower than source terminal then conducting path established between drain and source terminals (switch is closed)



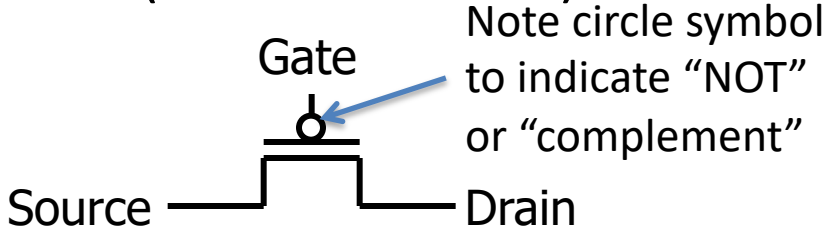
n-channel transitor

off when voltage at Gate is low

on when:

voltage (Gate) > voltage (Threshold)

(**High** resistance when gate voltage **Low**, **Low** resistance when gate voltage **High**)



p-channel transistor

on when voltage at Gate is low

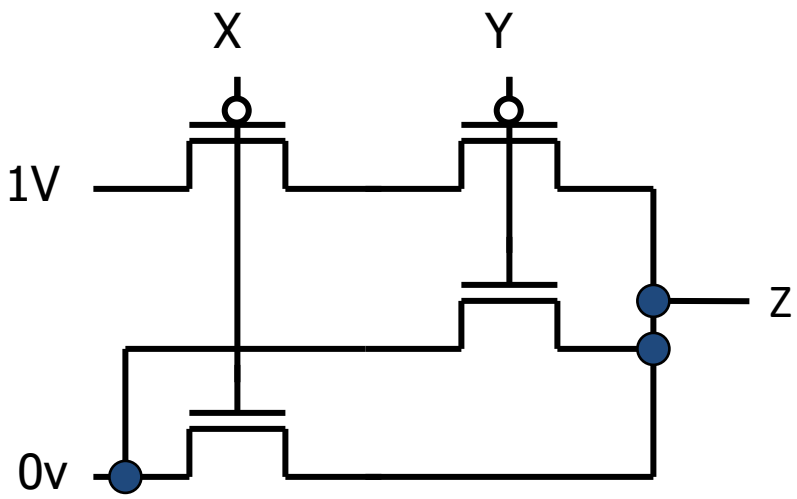
off when:

voltage (Gate) > voltage (Threshold)

(**Low** resistance when gate voltage **Low**, **High** resistance when gate voltage **High**)

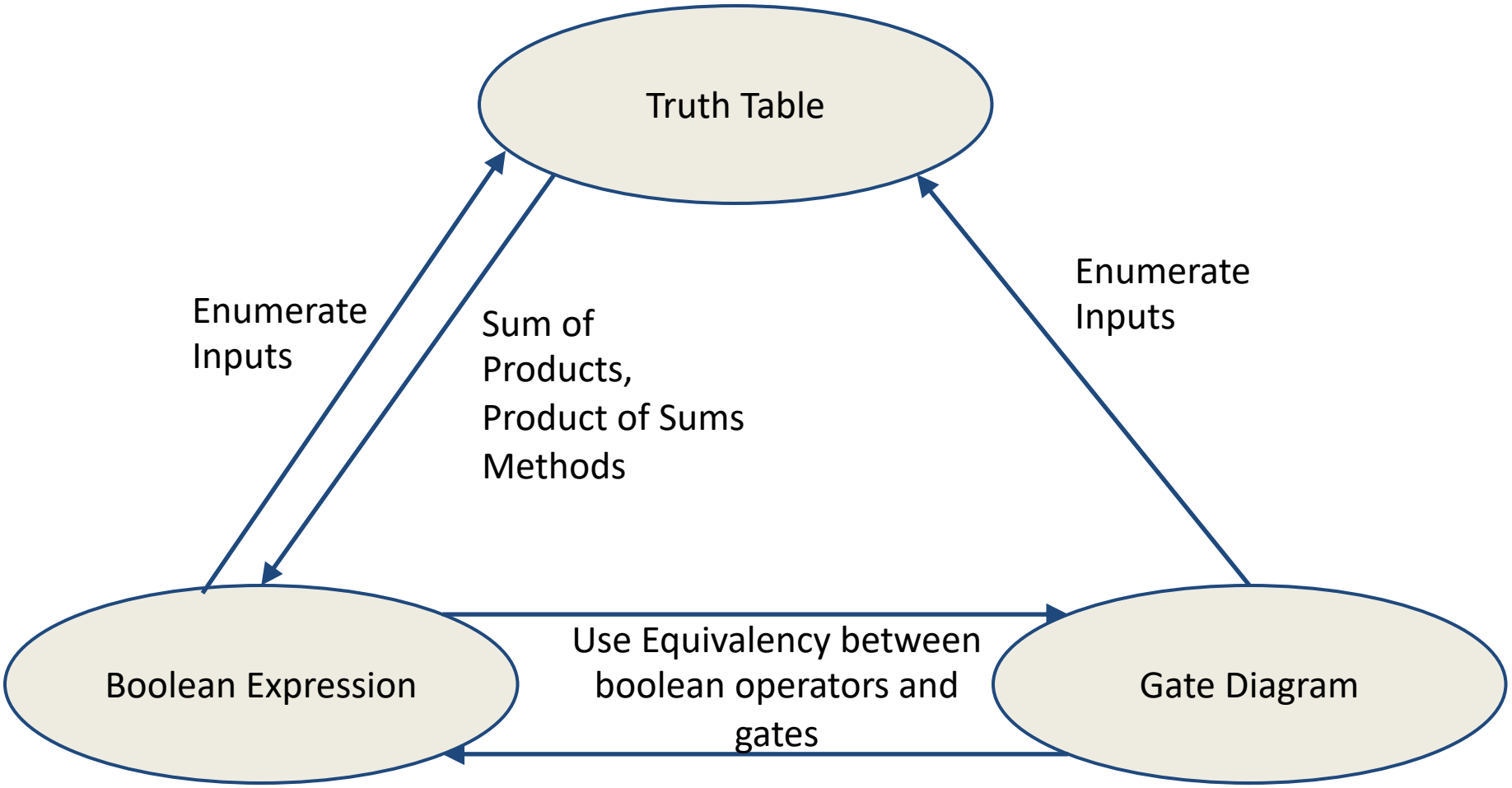
Field-Effect Transistor (FET) => CMOS circuits use a combination of p-type and n-type metal–oxide–semiconductor field-effect transistors => MOSFET

Question



X	Y	Z				
		A	B	C	D	
0 Volt	0 Volt	0	0	1	1	Volts
0 Volt	1 Volt	0	1	0	1	Volts
1 Volt	0 Volt	0	1	0	1	Volts
1 Volt	1 Volt	1	1	0	0	Volts

Representations of Combinational Logic (groups of logic gates)

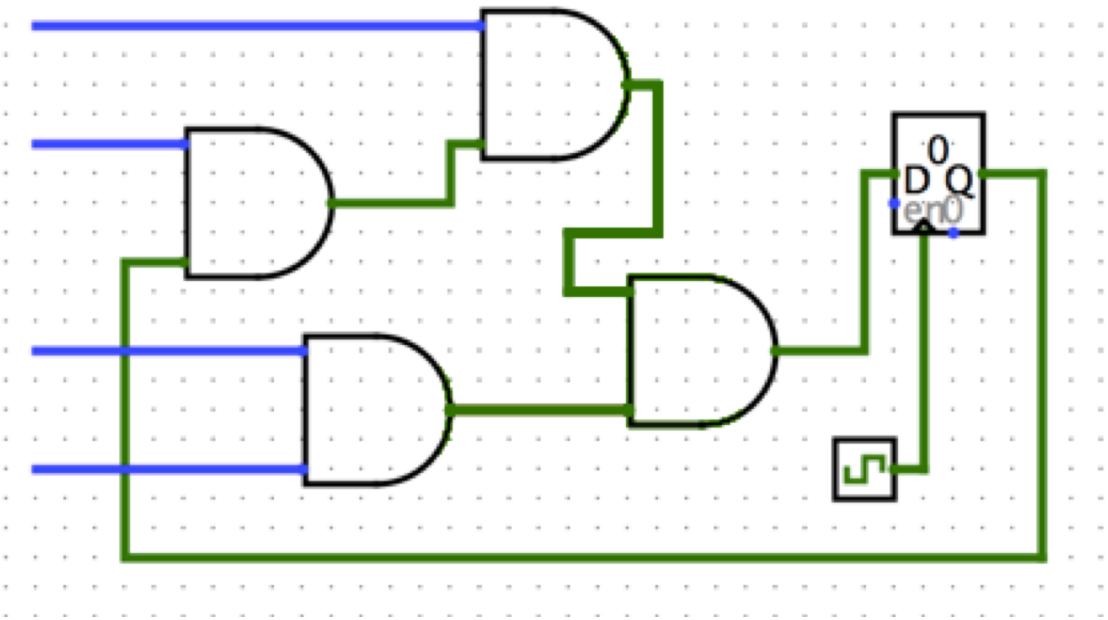


Laws of Boolean Algebra

$X \overline{X} = 0$	$X + \overline{X} = 1$	Complementarity
$X\ 0 = 0$	$X + 1 = 1$	Laws of 0's and 1's
$X\ 1 = X$	$X + 0 = X$	Identities
$X\ X = X$	$X + X = X$	Idempotent Laws
$X\ Y = Y\ X$	$X + Y = Y + X$	Commutativity
$(X\ Y)\ Z = X\ (Y\ Z)$	$(X + Y) + Z = X + (Y + Z)$	Associativity
$X\ (Y + Z) = X\ Y + X\ Z$	$X + Y\ Z = (X + Y)\ (X + Z)$	Distribution
$X\ Y + X = X$	$(X + Y)\ X = X$	Uniting Theorem
$\overline{X}\ Y + X = X + Y$	$(\overline{X} + Y)\ X = X\ Y$	Uniting Theorem v. 2
$\overline{X\ Y} = \overline{X} + \overline{Y}$	$\overline{X + Y} = \overline{X}\ \overline{Y}$	DeMorgan's Law

Question

Piazza: "Lecture 9 Freq Poll"



Clock->Q 1ns
Setup 1ns
Hold 1ns
AND delay 1ns

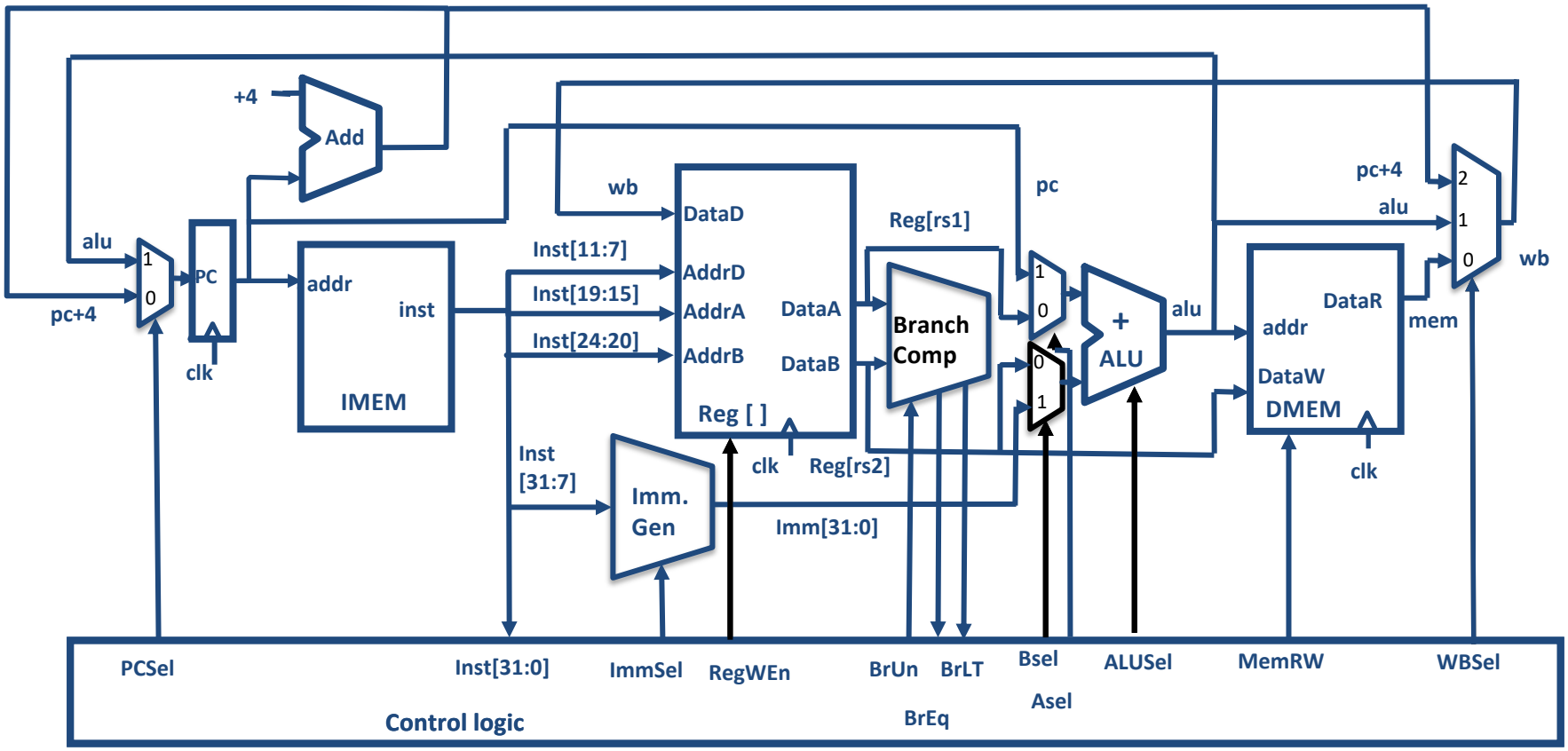
What is maximum clock frequency?

- A: 5 GHz
- B: 500 MHz
- C: 200 MHz
- D: 250 MHz
- E: 1/6 GHz

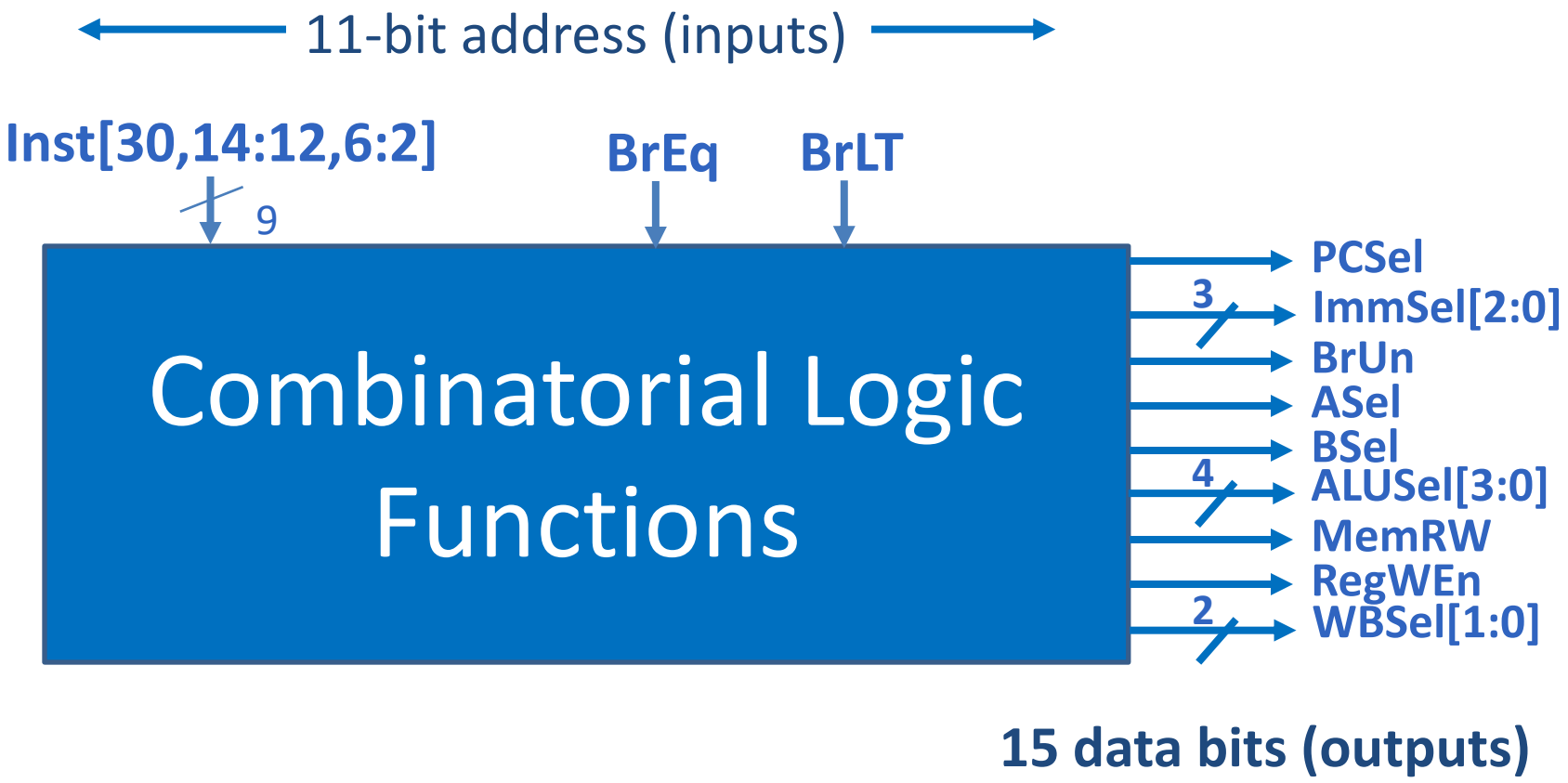
Keywords

- Mux
- Register
- FlipFlop
- Adder
- ALU
- AND; OR; XOR; NOT
- NAND; NOR

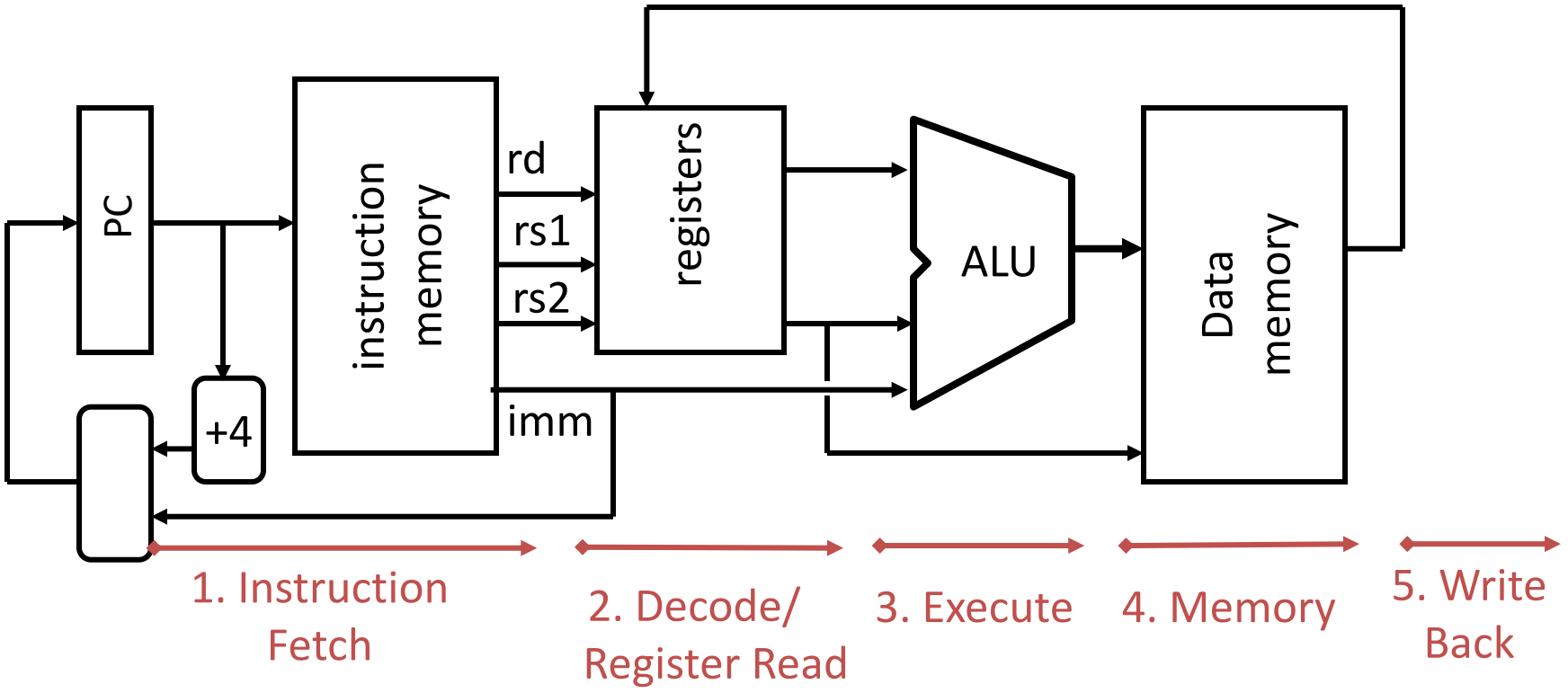
Complete RV32I Datapath!



Control Block Design



Single Cycle Datapath



Pipelining with RISC-V

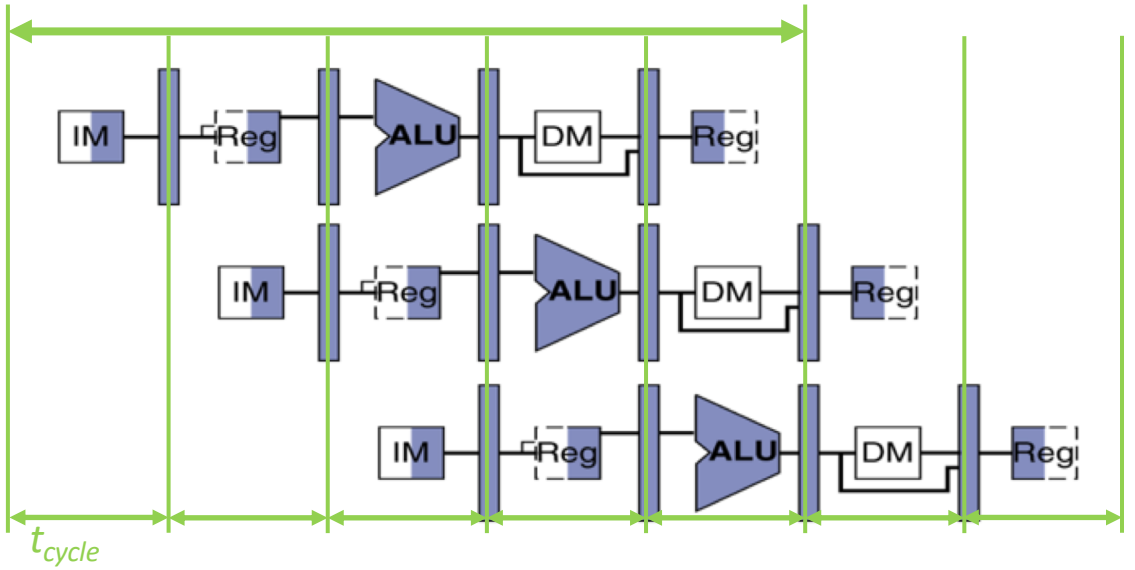
Phase	Pictogram	t_{step} Serial	t_{cycle} Pipelined
Instruction Fetch	IM	200 ps	200 ps
Reg Read	Reg	100 ps	200 ps
ALU	ALU	200 ps	200 ps
Memory	DM	200 ps	200 ps
Register Write	Reg	100 ps	200 ps
$t_{instruction}$	IM Reg ALU DM Reg	**800 ps**	**1000 ps**

instruction sequence

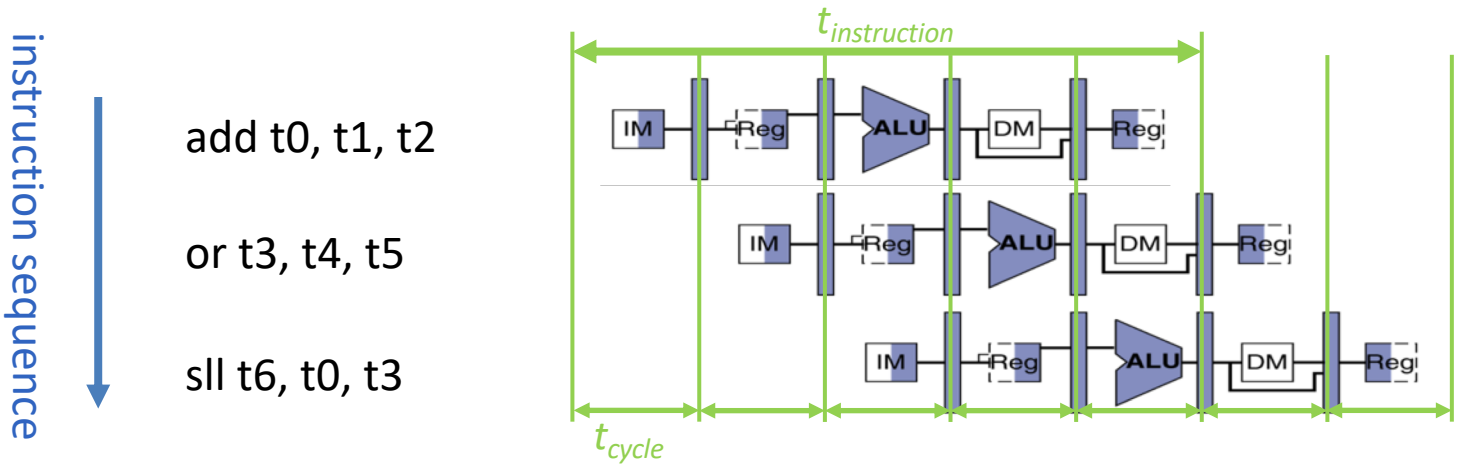
add t0, t1, t2

or t3, t4, t5

sll t6, t0, t3



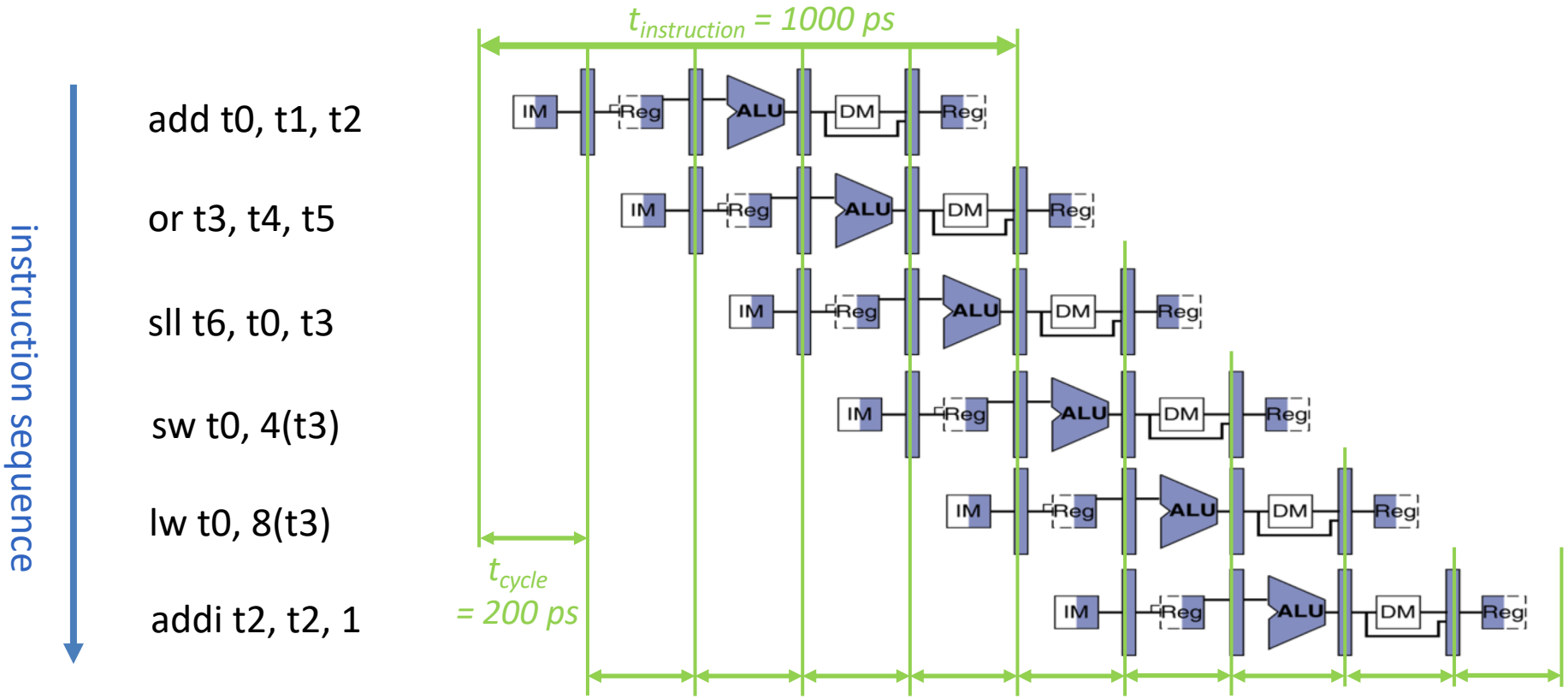
Pipelining with RISC-V



	Single Cycle	Pipelining
Timing	t_{step} = 100 … 200 ps	t_{cycle} = 200 ps
	Register access only 100 ps	All cycles same length
Instruction time, $t_{instruction}$	= t_{cycle} = 800 ps	1000 ps
CPI (Cycles Per Instruction)	~1 (ideal)	~1 (ideal), >1 (actual)
Clock rate, f_s	1/800 ps = 1.25 GHz	1/200 ps = 5 GHz
Relative speed	1 x	4 x

Sequential vs Simultaneous

What happens sequentially, what happens simultaneously?



RISC-V Pipeline

instruction sequence

add t0, t1, t2

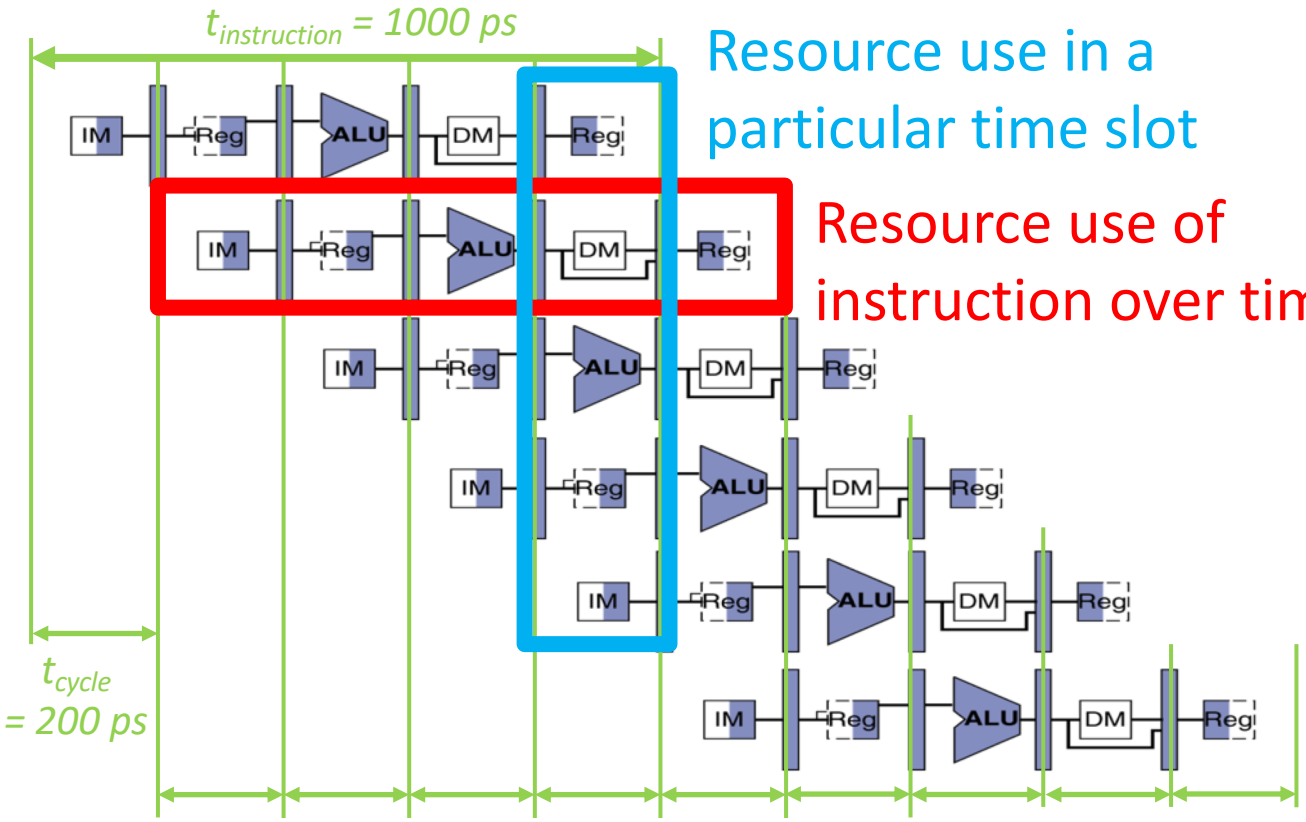
or t3, t4, t5

sll t6, t0, t3

sw t0, 4(t3)

lw t0, 8(t3)

addi t2, t2, 1

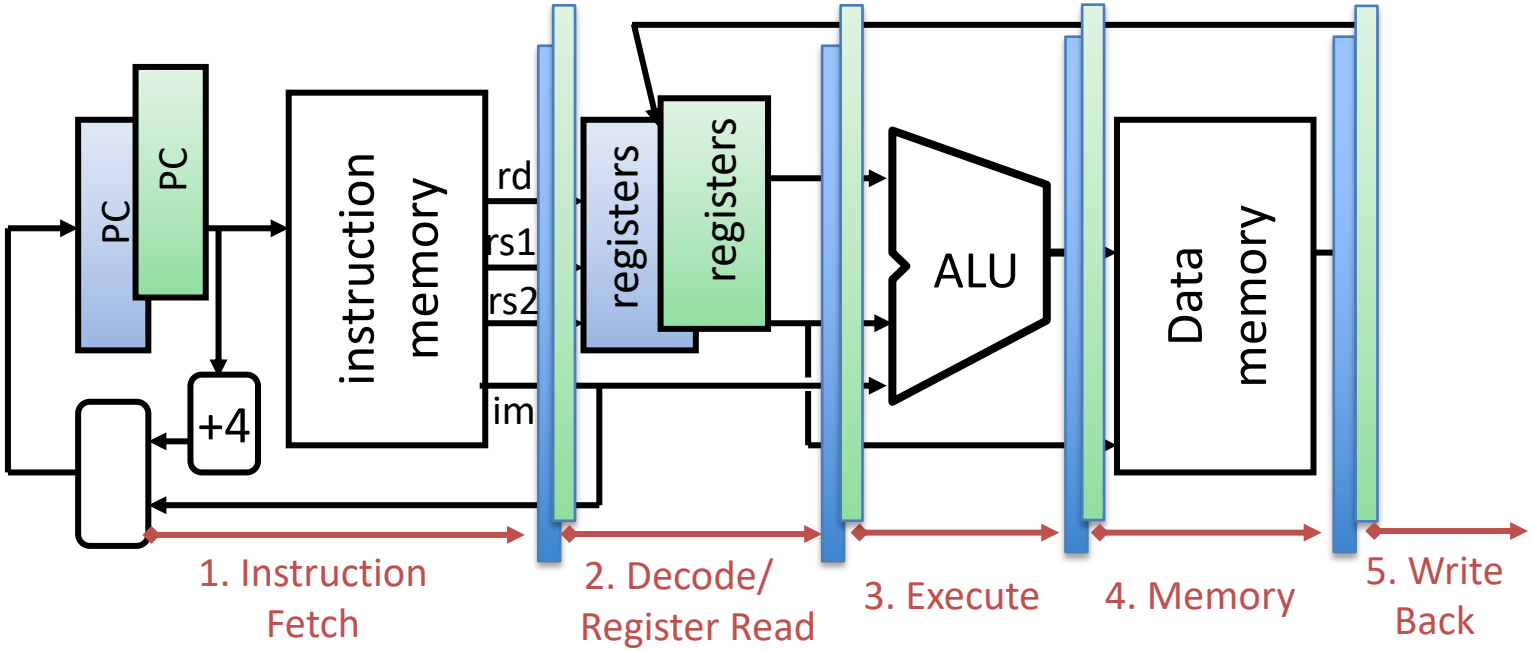


Hazards!

- Keywords:
  - Structural Hazards
  - Data Hazards
  - Control Hazards

  - Forwarding/ Bypassing
  - Stall/ nop/ load delay slot
  - Kill instruction
  - Branch prediciton

Hyper-threading (simplified)

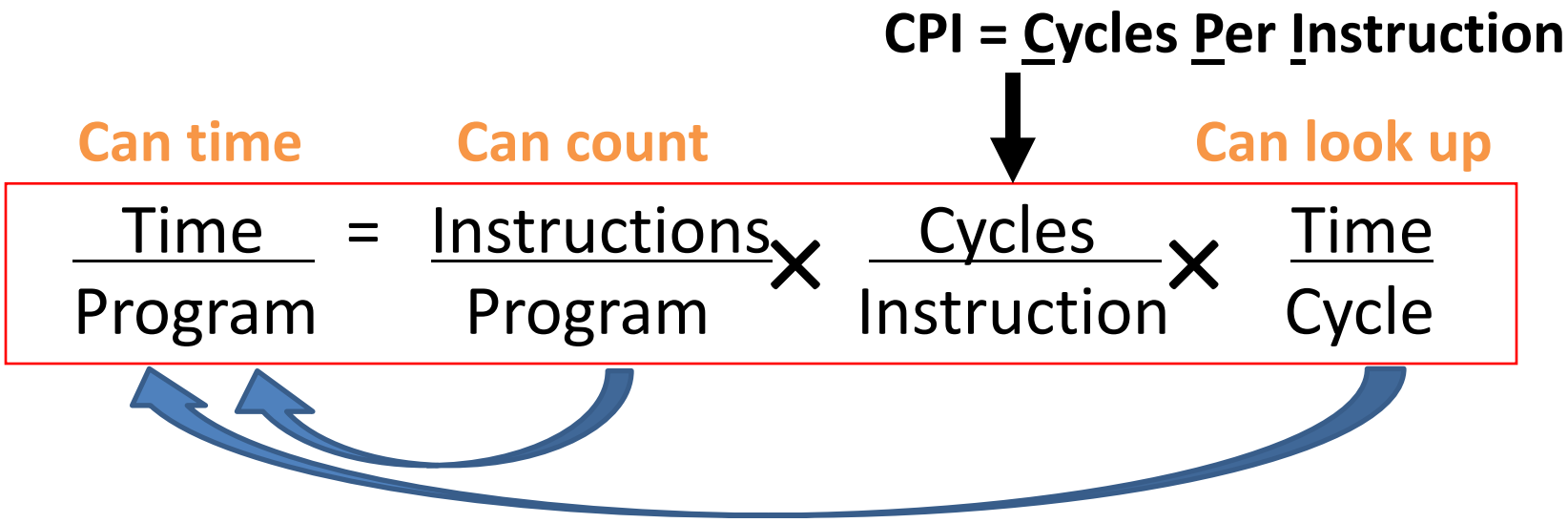


- Duplicate all elements that hold the state (registers)
- Use the same CL blocks
- Use muxes to select which state to use every clock cycle
- => run 2 independent processes
  - No Hazards: registers different; different control flow; memory different; Threads: memory hazard should be solved by software (locking, mutex, …)
- Speedup?
  - No obvious speedup; Complex pipeline: make use of CL blocks in case of unavailable resources (e.g. wait for memory)

Superscalar

- "Iron Law" of Processor Performance to estimate speed
- Complex Pipelines
    - Multiple Functional Units => Parallel execution
  - Static Multiple Issues (VLIW)
    - E.g. 2 instructions per cycle
  - Dynamic Multiple Issues (Superscalar)
    - Re-order instructions
    - Issue Buffer; Re-order Buffer; Commit Unit
    - Re-naming of registeres

"Iron Law" of Processor Performance



$$\textbf{CPI} = \frac{\text{Cycles}}{\text{Instruction}} = \frac{\text{Time}}{\text{Program}} \div \left(\frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Time}}{\text{Cycle}} \right)$$

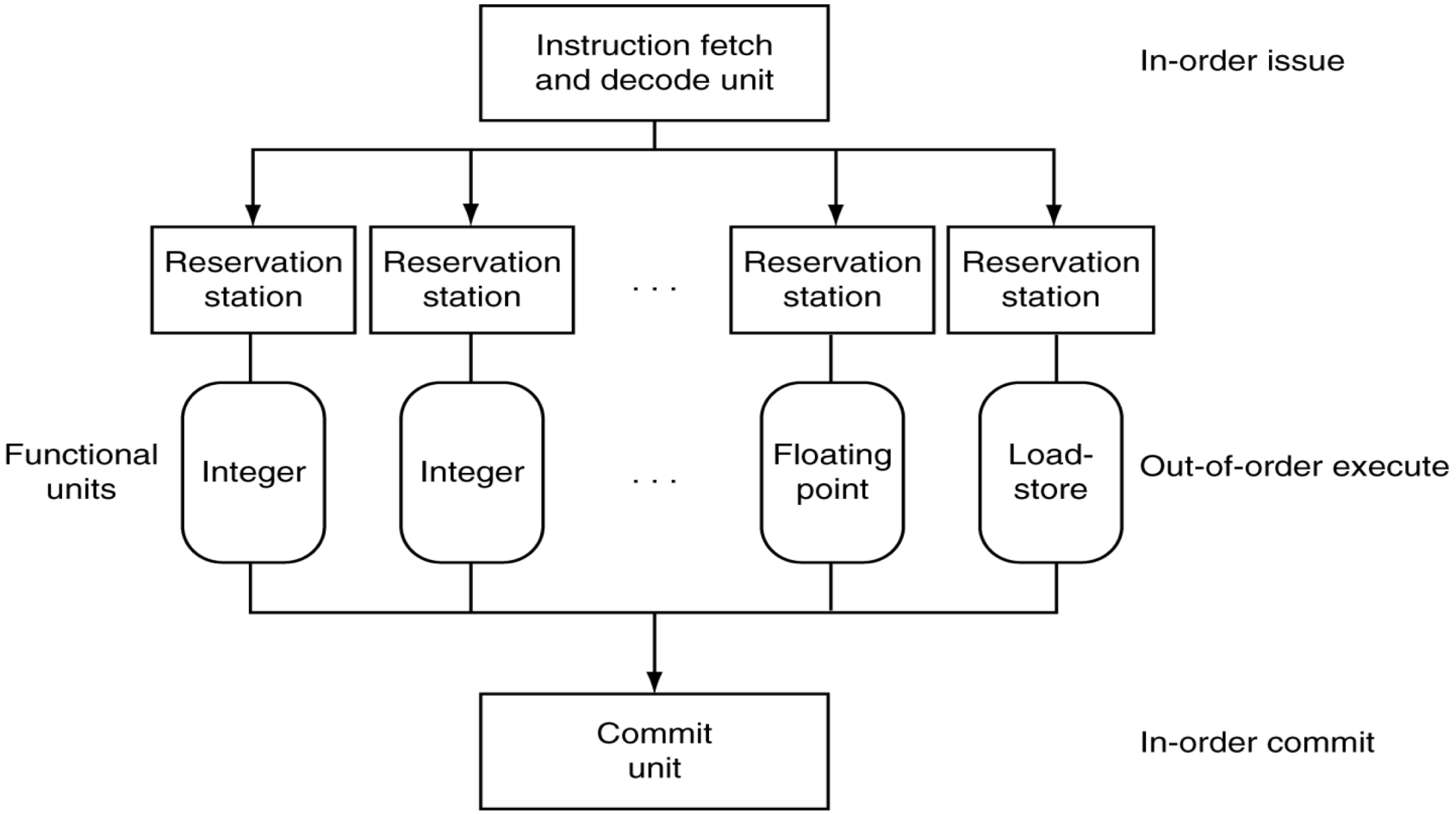
Example (RISC processor)

Op	$Freq_i$	CPI_i	Prod	(% Time)
ALU	50%	1	.5	(23%)
Load	20%	5	1.0	(45%)
Store	10%	3	.3	(14%)
Branch	20%	2	.4	(18%)
			2.2	

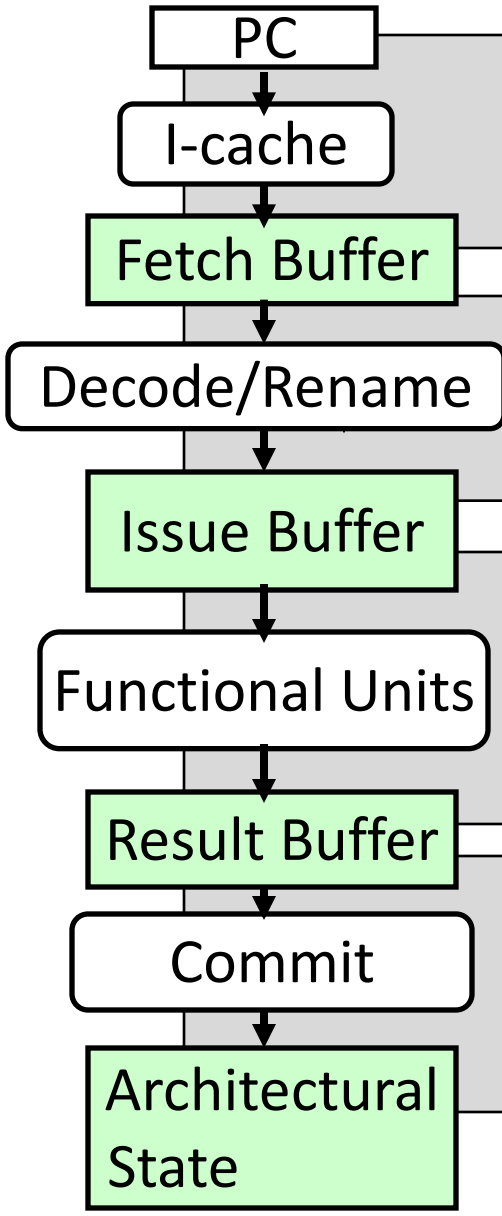
Instruction Mix

(Where time spent)

Superscalar Processor



Phases of Instruction Execution



Fetch: *Instruction bits retrieved from instruction cache.*

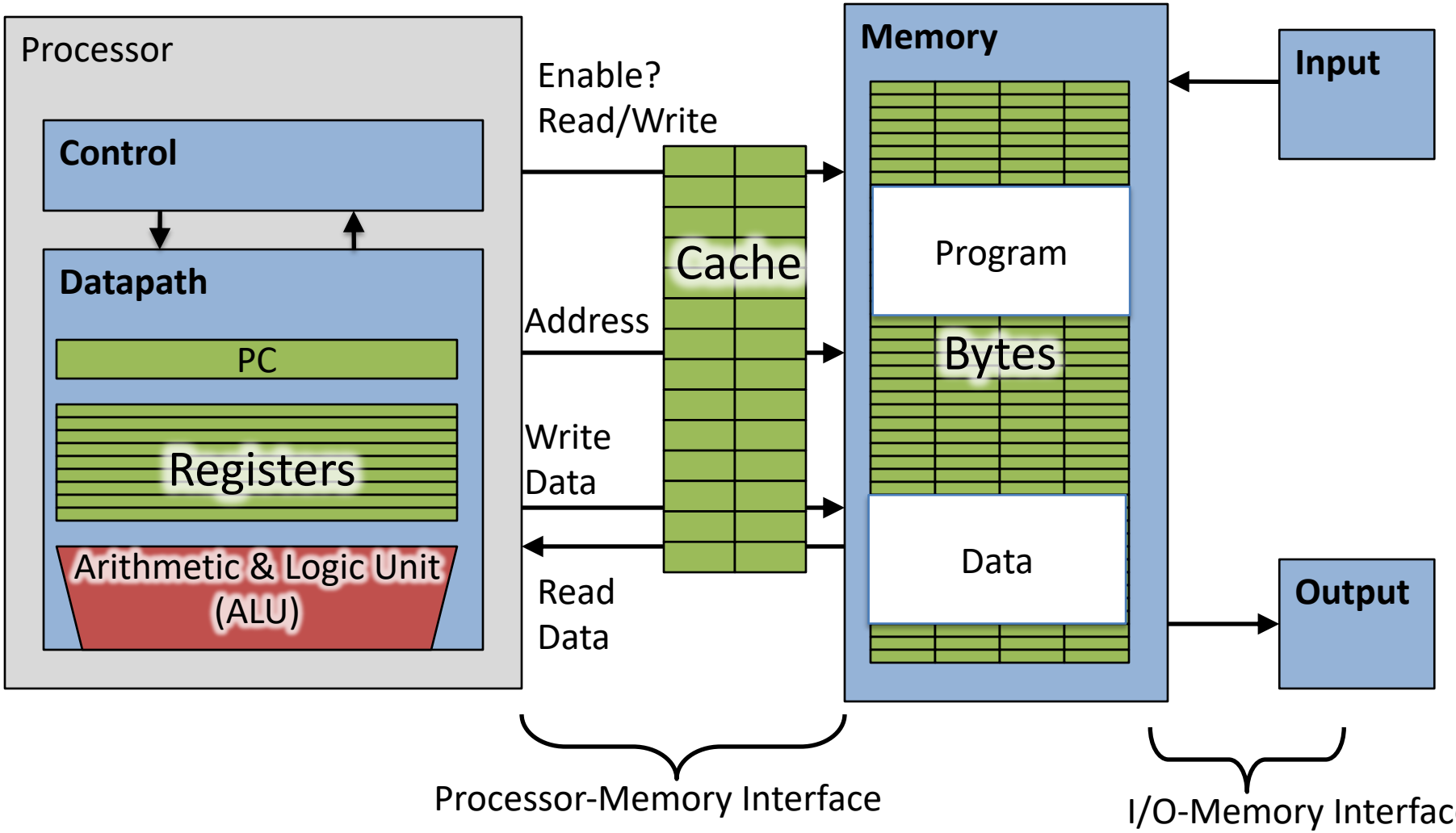
Decode: *Instructions dispatched to appropriate issue buffer*

Execute: *Instructions and operands issued to functional units. When execution completes, all results and exception flags are available.*

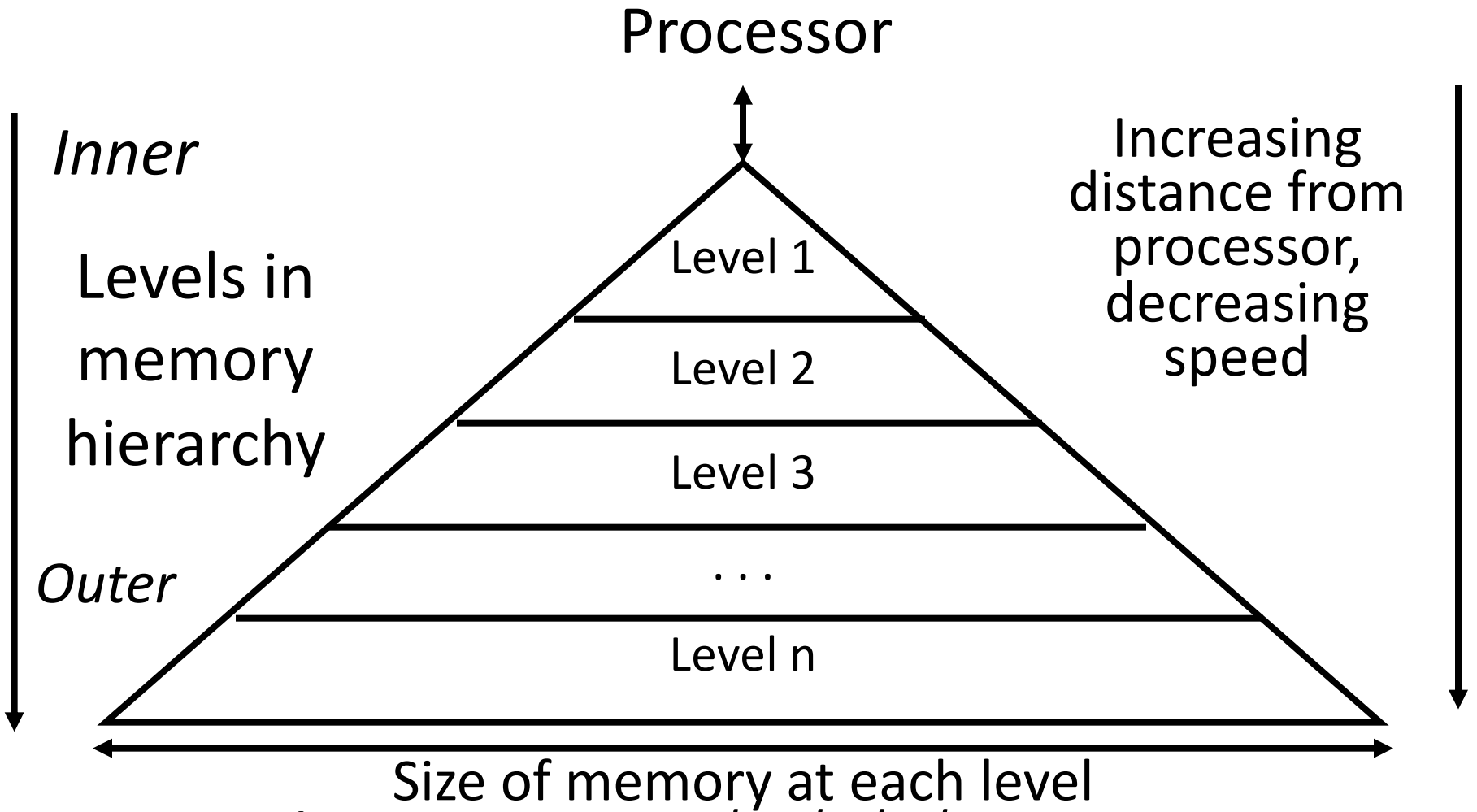
Commit: *Instruction irrevocably updates architectural state (aka "graduation"), or takes precise trap/interrupt.*

Intro to Caches

Adding Cache to Computer



Big Idea: Memory Hierarchy

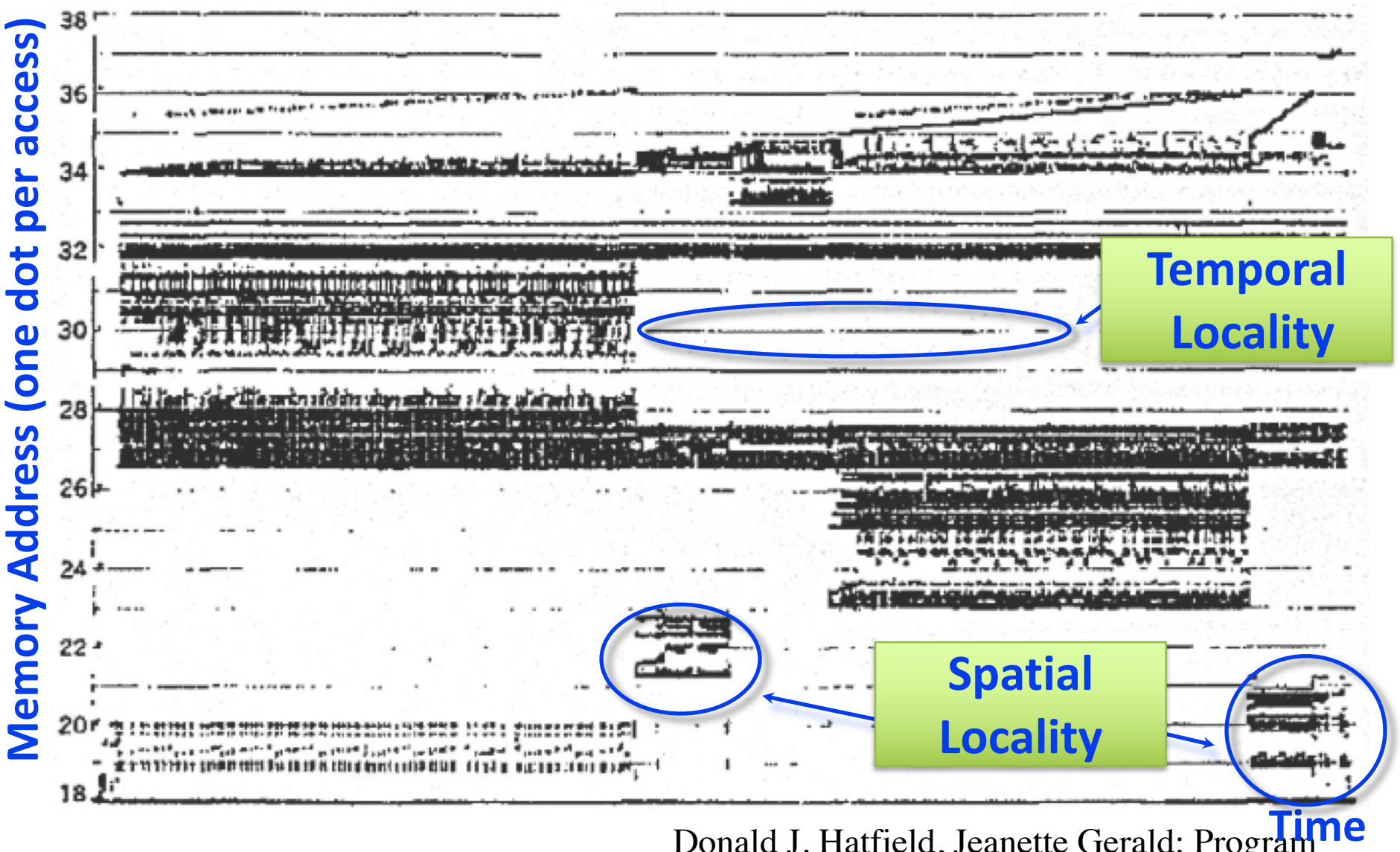


As we move to outer levels the latency goes up and price per bit goes down. Why?

Big Idea: Locality

- *Temporal Locality* (locality in time)
  - Go back to same book on desktop multiple times
  - If a memory location is referenced, then it will tend to be referenced again soon
- *Spatial Locality* (locality in space)
  - When go to book shelf, pick up multiple books on J.D. Salinger since library stores related books together
  - If a memory location is referenced, the locations with nearby addresses will tend to be referenced soon

Memory Reference Patterns



Donald J. Hatfield, Jeanette Gerald: Program Restructuring for Virtual Memory. IBM Systems Journal 10(3): 168-192 (1971)

Principle of Locality

- *Principle of Locality*: Programs access small portion of address space at any instant of time (spatial locality) and repeatedly access that portion (temporal locality)
- What program structures lead to temporal and spatial locality in **instruction** accesses?
- In **data** accesses?

Cache Philosophy

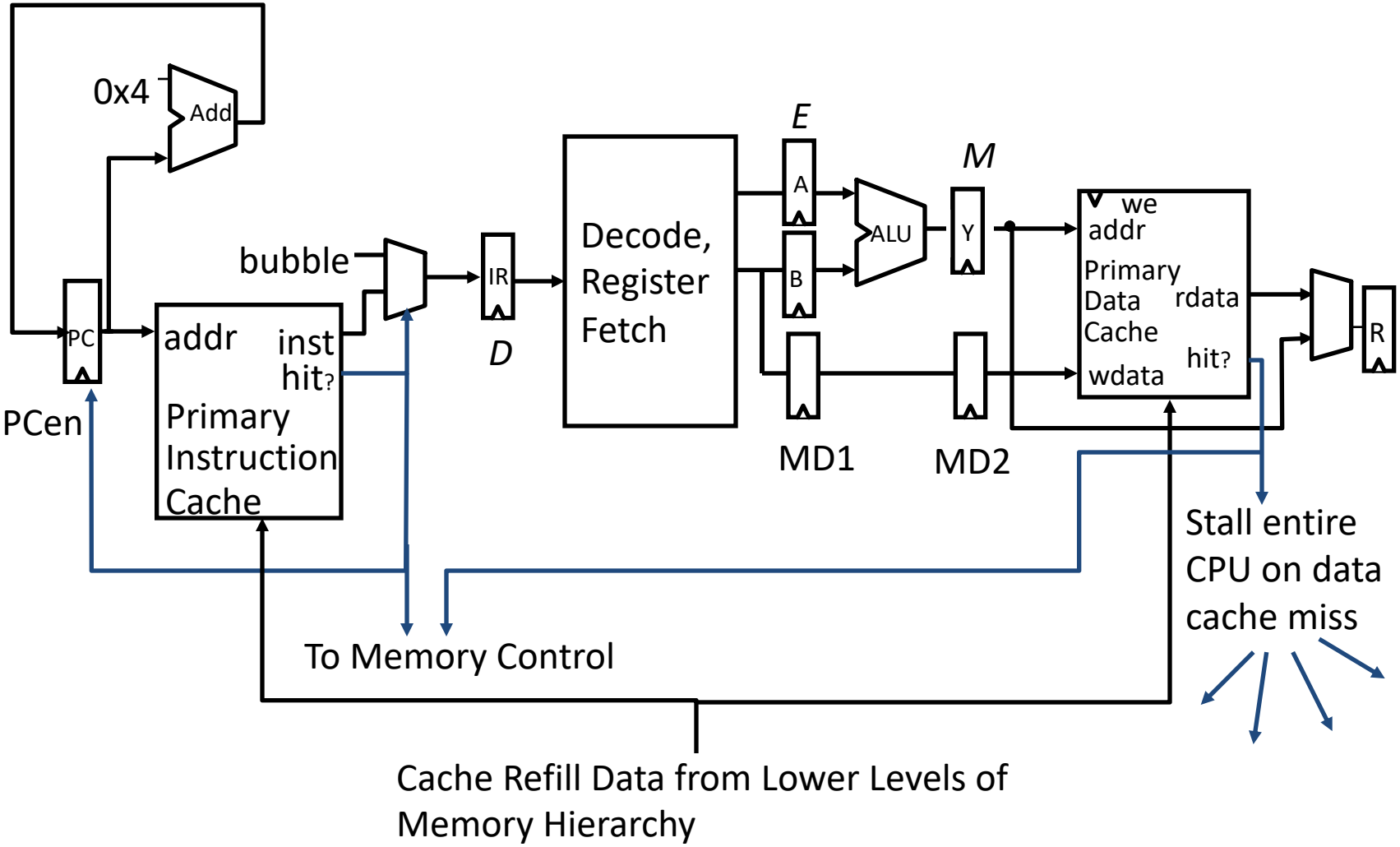
- Programmer-invisible hardware mechanism to give illusion of speed of fastest memory with size of largest memory
  - Works fine even if programmer has no idea what a cache is
  - However, performance-oriented programmers today sometimes “reverse engineer” cache design to design data structures to match cache

Cache Terms I

- ***Cache***:
  - A small and fast memory used to increase the performance of accessing a big and slow memory
  - Uses ***temporal locality:*** The tendency to reuse data in the same space over time
  - Uses ***spacial locality:*** The tendency to use data at addresses near
- Cache ***hit***: The address being fetched is in the cache
- Cache ***miss:*** The address being fetched is not in the cache
- ***Valid bit***: Is a particular entry valid
- Cache ***flush***: Invalidate all entries

CPU-Cache Interaction

(5-stage pipeline)



Cache Terms II

- Cache ***level***:
  - The order in the memory hierarchy: L1$ is closest to the processor
  - L1 caches may only hold data (Data-cache, D$) or instructions (Instruction Cache, I$)
    - Most L2+ caches are "unified", can hold both instructions and data
- Cache ***capacity***:
  - The total # of bytes in the cache
- Cache ***line*** or cache ***block***:
  - A single entry in the cache
- Cache ***block size***:
  - The number of bytes in each cache line

Cache Terms III
Associativity

- ***Number of cache lines***:
  - Cache capacity / block size:
- Cache ***associativity***:
  - The number of possible cache lines a given address may exist in.
  - Also the number of comparison operations needed to check for an element in the cache
  - ***Direct mapped***: A data element can only be in one possible location (N=1)
  - ***N-way set associative***: A data element can be in one of N possible positions
  - ***Fully associative***: A data element can be at any location in the cache.
    - Associativity == # of lines
- Total # of cache lines == capacity of cache/line size

- Total # of lines in a set == # ways == N == associativity
- Total # of sets == # of cache lines / associativity

Victim Cache

- Conflict misses are a pain, but...
  - Perhaps a little associativity can help without having to be a fully associative cache
- In addition to the main cache...
  - Optionally have a very small (16-64 entry) ***fully associative*** "victim" cache
- Whenever we evict a cache entry
  - Don't just get rid of it, put it in the victim cache
- Now on cache misses...
  - Check the victim cache first, if it is in the victim cache you can just reload it from there

Cache Terms IV
Parts of the Address

- Address is divided into |***TAG***|***INDEX***|***OFFSET***|
- ***Offset***:
  - The lowest bits of the memory address which say where data exists within the cache line.
  - It is log2(line/block size)
  - So for a cache with 64B blocks it is 6 bits
- ***Index***:
  - The portion of the address which says where in the cache an address may be stored
  - Takes log2(# of cache lines / associativity) bits
  - So for a 4 way associative cache with 512 lines it is 7 bits
- ***Tag***: The portion of the address which must be stored in the cache to check if a location matches
  - # of bits of address - (# of bits for index + # of bits for offset)
  - So with 64b addresses it is 51b...

Cache Terms V
Writing

- ***Eviction***:
  - The process of removing an entry from the cache
- ***Write Back***:
  - A cache which only writes data up the hierarchy when a cache line is evicted
  - Instead set a ***dirty bit*** on cache entries
  - All i7 caches are ***write back***
- ***Write Through***:
  - A cache which always writes to memory
- ***Write Allocate***:
  - If writing to memory ***not in the cache*** fetch it first
  - i7 L2 is Write Allocate
- ***No Write Allocate***:
  - Just write to memory without a fetch
  - i7 L1 is no write allocate

Replacement Policy

In an associative cache, which line from a set should be evicted when the set becomes full?

- Random
- Least-Recently Used (LRU)
  - LRU cache state must be updated on every access
  - True implementation only feasible for small sets (2-way)
  - Pseudo-LRU binary tree often used for 4-8 way
- First-In, First-Out (FIFO) a.k.a. Round-Robin
  - Used in highly associative caches
- Not-Most-Recently Used (NMRU)
  - FIFO with exception for most-recently used line or lines

This is a second-order effect. Why?

Replacement only happens on misses

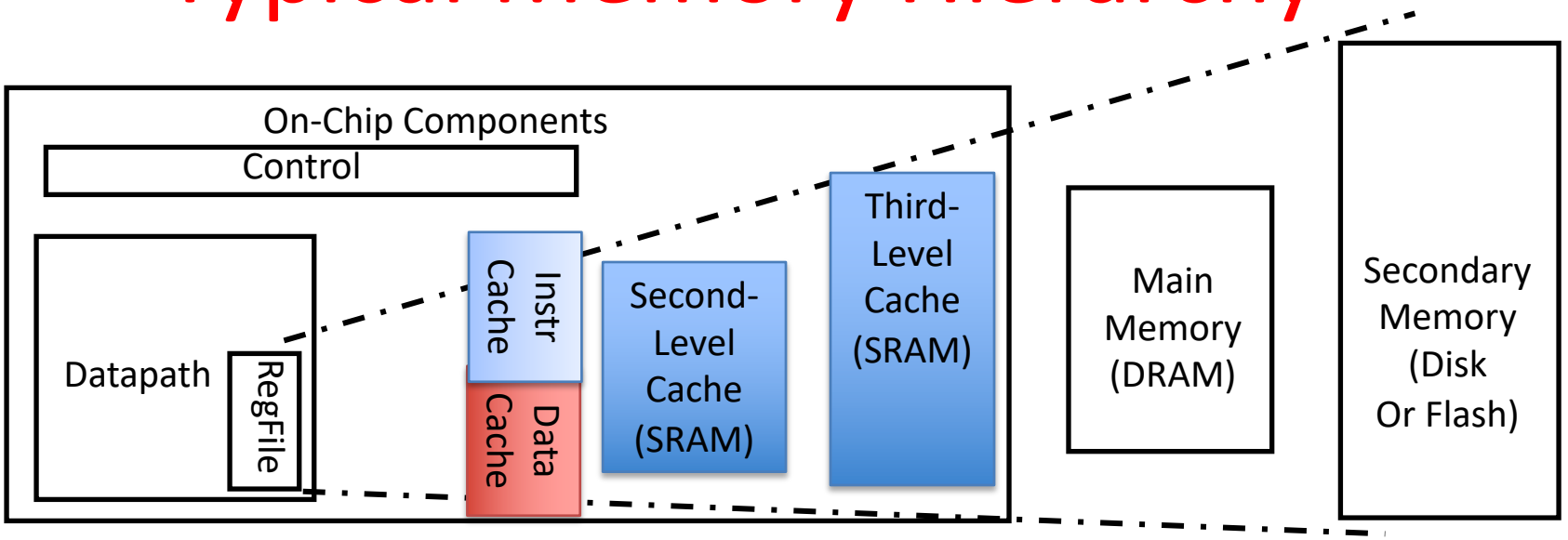
Cache Terms VI
Cache Performance

- ***Hit Time***:
  - Amount of time to return data in a given cache: depends on the cache
  - i7 L1 hit time: 4 clock cycles
- ***Miss Penalty***:
  - Amount of ***additional*** time to return an element if its not in the cache: depends on the cache
- ***Miss Rate***:
  - Fraction of a ***particular program's*** memory requests which miss in the cache
- Average Memory Access Time (***AMAT***):
  - Hit time + Miss Rate * Miss Penalty

Understanding Cache Misses: The 3Cs

- Compulsory (cold start or process migration, 1st reference):
  - First access to block impossible to avoid; small effect for long running programs
  - Solution: increase block size (increases miss penalty; very large blocks could increase miss rate)
- Capacity:
  - Cache cannot contain all blocks accessed by the program
  - Solution: increase cache size (may increase access time)
- *Conflict (collision):*
  - *Multiple memory locations mapped to the same cache location*
  - *Solution 1: increase cache size*
  - *Solution 2: increase associativity (may increase access time)*

Typical Memory Hierarchy



Speed (cycles):	½'s	1's	10's	100's	1,000,000's
Size (bytes):	100's	10K's	M's	G's	T's
Cost/bit:	highest	←		→	lowest

- Principle of locality + memory hierarchy presents programmer with ≈ as much memory as is available in the *cheapest* technology at the ≈ speed offered by the *fastest* technology

Local vs. Global Miss Rates

- *Local miss rate* – the fraction of references to one level of a cache that miss
- Local Miss rate L2$ = $L2 Misses / L1$ Misses
- *Global miss rate* – the fraction of references that miss in all levels of a multilevel cache
  - L2$ local miss rate >> than the global miss rate
- Global Miss rate = L2$ Misses / Total Accesses
  = (L2$ Misses / L1$ Misses) × (L1$ Misses / Total Accesses)
  = Local Miss rate L2$ × Local Miss rate L1$

- AMAT = Time for a hit + Miss rate × Miss penalty
- AMAT = Time for a L1$ hit + (local) Miss rate L1$ × (Time for a L2$ hit + (local) Miss rate L2$ × L2$ Miss penalty)

In Conclusion, Cache Design Space

- Several interacting dimensions
  - Cache size
  - Block size
  - Associativity
  - Replacement policy
  - Write-through vs. write-back
  - Write-allocation
- Optimal choice is a compromise
  - Depends on access characteristics
    - Workload
    - Use (I-cache, D-cache)
  - Depends on technology / cost
- Simplicity often wins