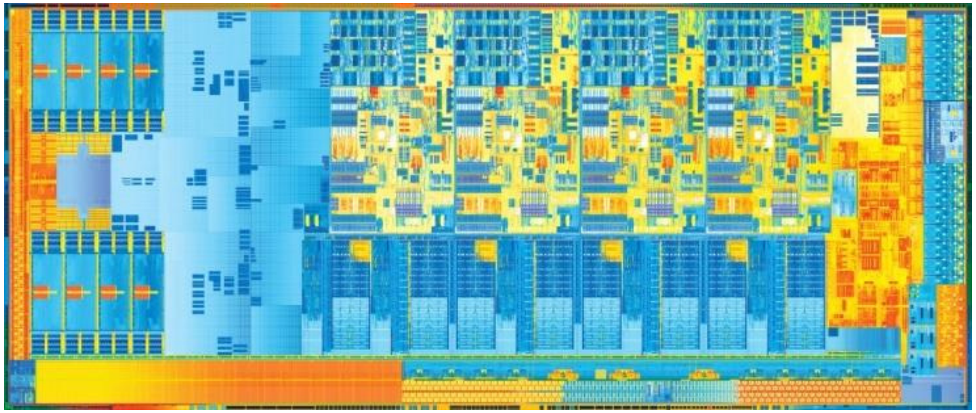
6.823 Computer System Architecture

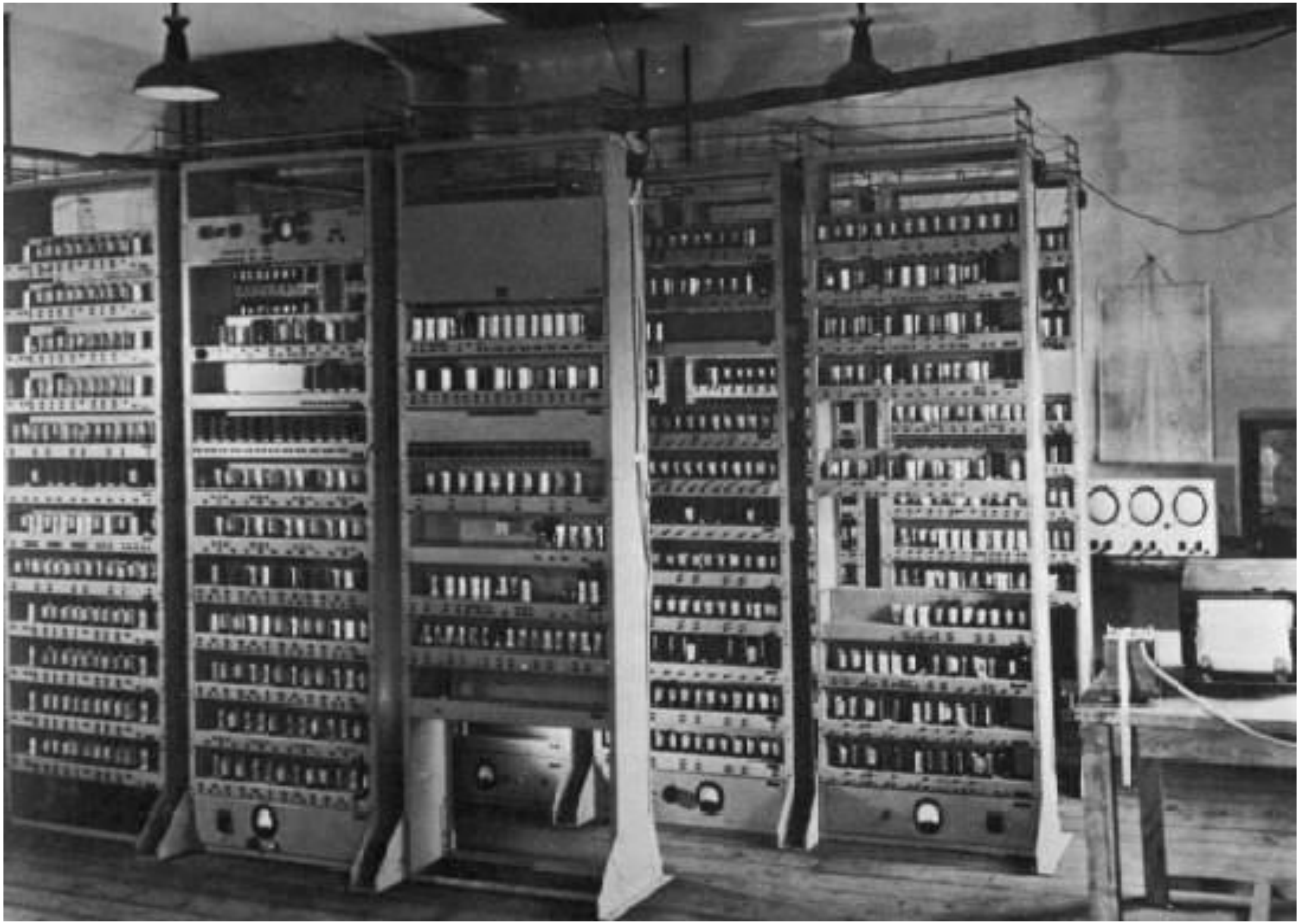
Lecturers: *Daniel Sanchez and Joel Emer*
TA: *Mark C. Jeffrey*

The processor you built in 6.004*

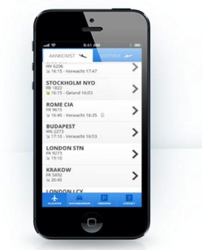


What you'll understand after taking 6.823

Computing devices then…



Computing devices now



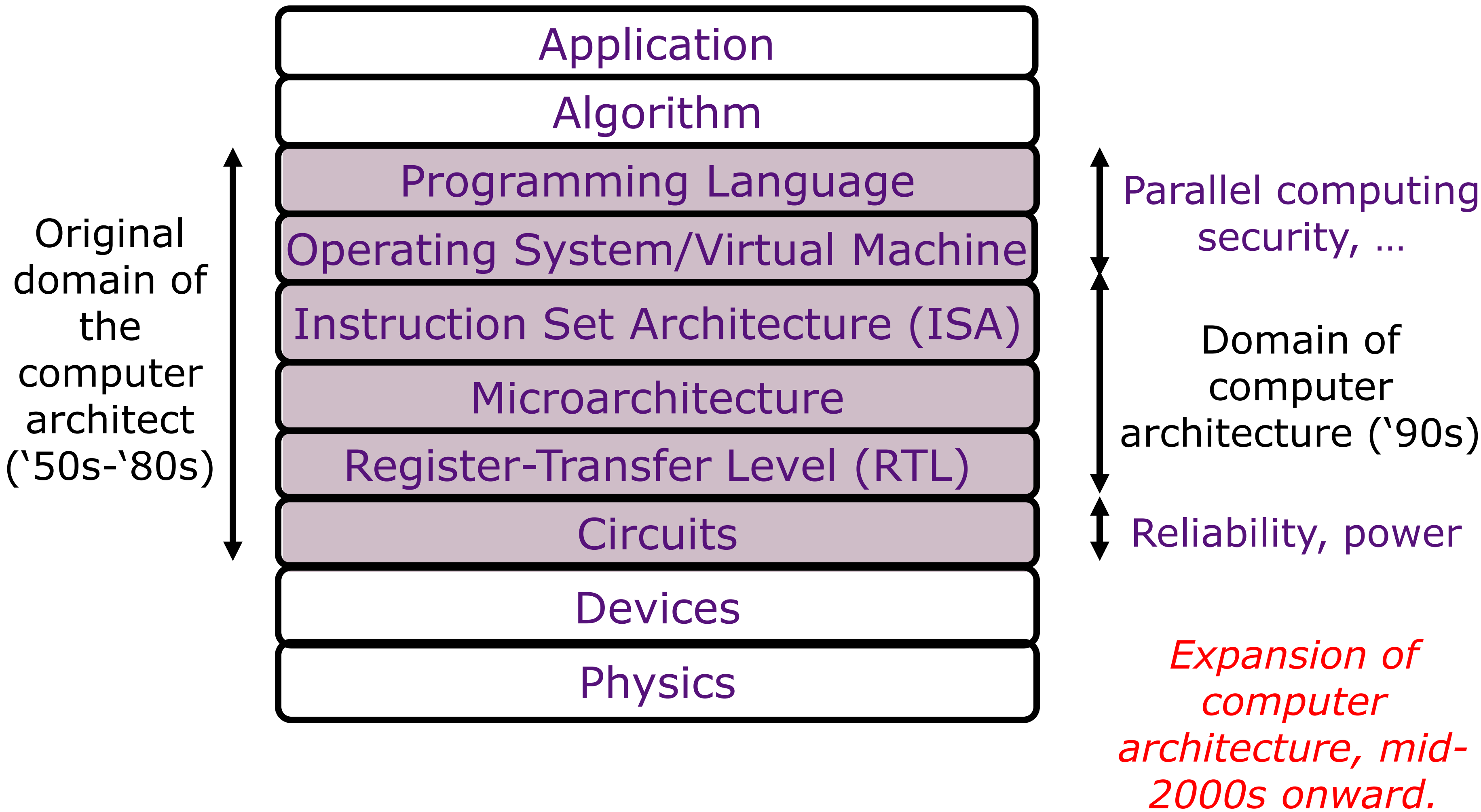





A journey through this space

- What do computer architects actually do?
- Illustrate via historical examples
  - Early days: ENIAC, EDVAC and EDSAC
  - Arrival of IBM 650 and then IBM 360
  - Seymour Cray – CDC 6600, Cray 1
  - Microprocessors and PCs
  - Multicores
  - Cell phones
- Focus on ideas, mechanisms and principles, especially those that have withstood the test of time

Abstraction layers



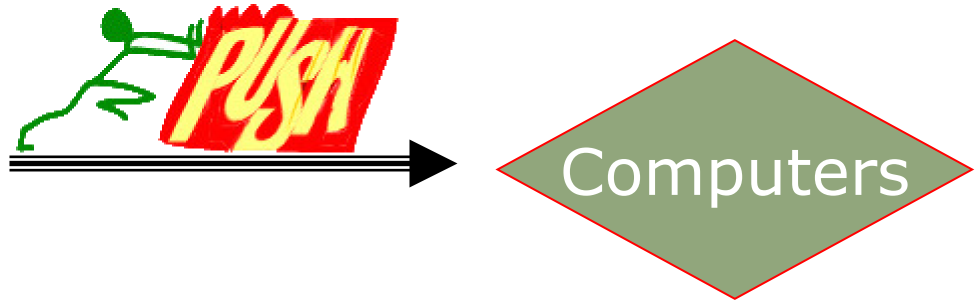
Computer Architecture is the design of abstraction layers

- What do abstraction layers provide?
  - Environmental stability within generation
  - Environmental stability across generations
  - Consistency across a large number of units

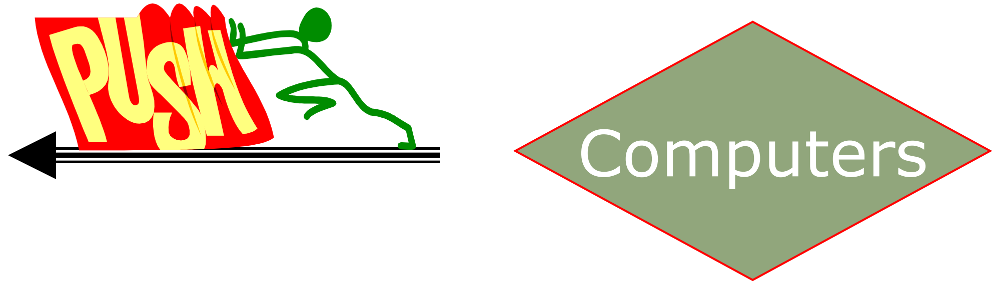
- What are the consequences?
  - *Encouragement to create reusable foundations:*
    - *Tool chains, operating systems, libraries*
  - Enticement for application innovation

Technology is the dominant factor in computer design

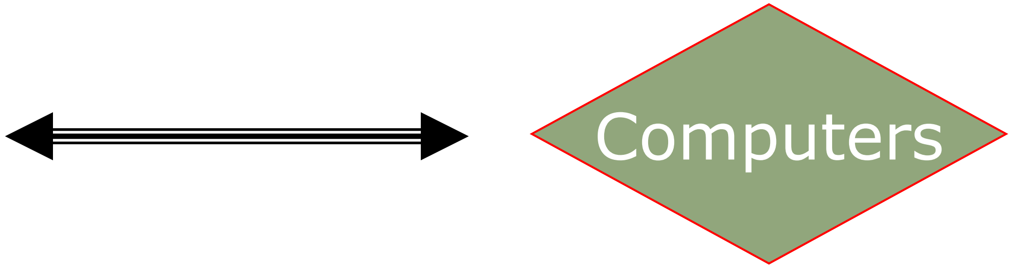
Technology
- *Transistors*
- *Integrated circuits*
- *VLSI (initially)*
- *Flash memories, …*



Technology
- *Core memories*
- *Magnetic tapes*
- *Disks*



Technology
- *ROMs, RAMs*
- *VLSI*
- *Packaging*
- *Low Power*

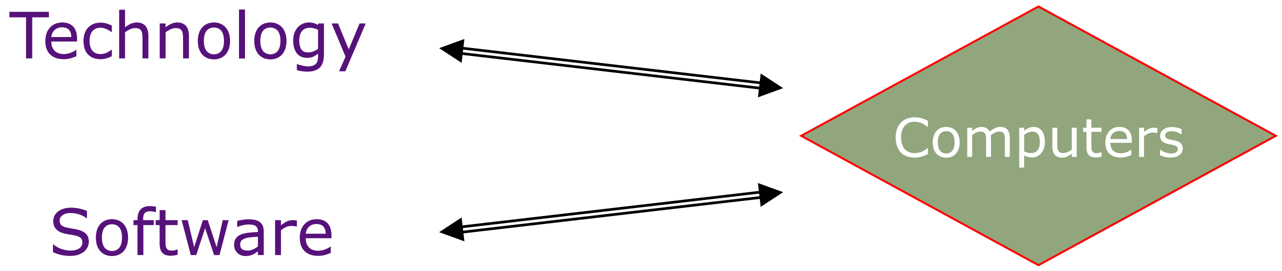


But Software…

As people write programs and use computers, our understanding of *programming* and *program behavior* improves.

This has profound though slower impact on computer architecture

Modern architects must pay attention to software and compilation issues.



Architecture is engineering design under constraints

Factors to consider:

- Performance of whole system on target applications
  - Average case & worst case
- Cost of manufacturing chips and supporting system
- Power to run system
  - Peak power & energy per operation
- Reliability of system
  - Soft errors & hard errors
- Cost to design chips (engineers, computers, CAD tools)
  - Becoming a limiting factor in many situations, fewer unique chips can be justified
- Cost to develop applications and system software
  - Often the dominant constraint for any programmable device

At different times, and for different applications at the same point in time, the relative balance of these factors can result in widely varying architectural choices

Course Information

All info kept up to date on the website:

`http://www.csg.csail.mit.edu/6.823`

Contact times

- Lectures Mondays and Wednesdays
  - 1:00pm to 2:30pm in room 32-141
- Tutorial on Fridays
  - 1:00pm to 2:00pm in room 37-212
  - Attendance is optional
  - Additional tutorials will be held in evenings before quizzes
- *Quizzes on Friday (except last quiz)*
  - 1:00pm to 2:30pm in room 37-212
  - Attendance is NOT optional
- Instructor office hours
  - After class or by email appointment
- TA office hours
  - Wednesday 4-5:30pm @ Stata 32G-725

The course has four modules

Module 1
- Instruction Set Architecture (ISA)
- Caches and Virtual Memory
- Simple Pipelining and Hazards

Module 2
- Complex Pipelining and Out of Order Execution
- Branch Prediction and Speculative Execution

Module 3
- Multithreading and Multiprocessors
- Coherence and consistency
- On-chip networks

Module 4
- VLIW, EPIC
- Vector machines and GPUs

Textbook and readings

- "Computer Architecture: A Quantitative Approach", Hennessy & Patterson, 5^{th} ed.
  - In reserve & available online through MIT Libraries
  - Recommended, but not necessary

- Course website lists H&P reading material for each lecture, and optional readings that provide more in-depth coverage

Grading

- Grades are not assigned based on a predetermined curve
  - Most of you are capable of getting an A
- 75% of the grade is based on four closed book 1.5 hour quizzes
  - The first three quizzes will be held during the tutorials; the last one during the last lecture (dates on web syllabus)
- 25% of the grade is based on four laboratory exercises
- No final exam
- No final projects
  - Take 6.175 next term if you're interested in building some of these machines!

Problem Sets & Labs

- Problem Sets
  - One problem set per module, not graded
  - Intended for private study and for tutorials to help prepare for quizzes
  - Quizzes assume you are very familiar with the content of problem sets

- Labs
  - Four graded labs (Lab 0 is introductory)
  - Based on widely-used PIN tool
  - Labs 2 and 4 are open-ended challenges

Self evaluation take-home quiz

- Goal is to help you judge for yourself whether you have prerequisites for this class, and to help refresh your memory
- We assume that you understand digital logic, a simple 5-stage pipeline, and simple caches
- Please work by yourself on this quiz – not in groups
- Remember to complete self-evaluation section at end of the quiz
- Due at start of class next Monday

Please email us if you have concerns about your ability to take the class

Early Developments: From ENIAC to the mid 50's

Prehistory

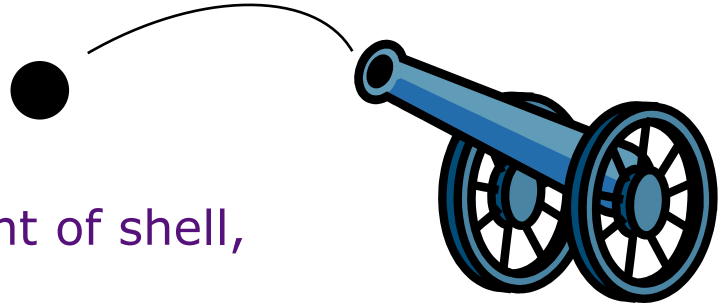
- 1800s: Charles Babbage
  - Difference Engine (conceived in 1823, first implemented in 1855 by Scheutz)
  - Analytic Engine, the first conception of a general purpose computer (1833, never implemented)
- 1890: Tabulating machines
- Early 1900s: Analog computers
- 1930s: Early electronic (fixed-function) digital computers

Electronic Numerical Integrator and Computer (ENIAC)

- Designed and built by Eckert and Mauchly at the University of Pennsylvania during 1943-45
- The first, completely electronic, operational, general-purpose analytical calculator!
  - 30 tons, 72 square meters, 200KW
- Performance
  - Read in 120 cards per minute
  - Addition took 200 μs, Division 6 ms
- Not very reliable!

WW-2 Effort

Application: Ballistic calculations



angle = f (location, tail wind, cross wind,
air density, temperature, weight of shell,
propellant charge, ...)

Electronic Discrete Variable Automatic Computer (EDVAC)

- ENIAC's programming system was external
  - Sequences of instructions were executed independently of the results of the calculation
  - Human intervention required to take instructions "out of order"
- EDVAC was designed by Eckert, Mauchly and von Neumann in 1944 to solve this problem
  - Solution was the *stored program computer*

    ⇒ *"program can be manipulated as data"*
- *First Draft of a report on EDVAC* was published in 1945, but just had von Neumann's signature!
  - Without a doubt the most influential paper in computer architecture

Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?

manual control	calculators
automatic control	
external (paper tape)	Harvard Mark I , 1944 Zuse's Z1, WW2
internal	
plug board	ENIAC 1946
read-only memory	ENIAC 1948
read-write memory	EDVAC 1947 *(concept)*

– The same storage can be used to store program and data

EDSAC	1950	Maurice Wilkes

The Spread of Ideas

ENIAC & EDVAC had immediate impact

brilliant engineering: Eckert & Mauchly
lucid paper: Burks, Goldstein & von Neumann

IAS	Princeton	46-52	Bigelow
EDSAC	Cambridge	46-50	Wilkes
MANIAC	Los Alamos	49-52	Metropolis
JOHNIAC	Rand	50-53	
ILLIAC	Illinois	49-52	
	Argonne	49-53	
SWAC	UCLA-NBS		

UNIVAC - the first commercial computer, 1951

Alan Turing's direct influence on these developments is often debated by historians.

Dominant Technology Issue: *Reliability*

ENIAC	⇒	EDVAC
18,000 tubes		4,000 tubes
20 10-digit numbers		2000 word storage
		mercury delay lines

Mean time between failures (MTBF)

MIT's Whirlwind with an MTBF of 20 min. was perhaps the most reliable machine!

Reasons for unreliability:

1. Vacuum tubes

2. Storage medium
   - Acoustic delay lines
   - Mercury delay lines
   - Williams tubes
   - Selections

CORE J. Forrester 1954

Computers in the mid 50's

- Hardware was expensive
- Stores were small (1000 words)

  ⇒ No resident system-software!

- Memory access time was 10 to 50 times slower than the processor cycle

  ⇒ Instruction execution time was totally dominated by the *memory reference time*.

- The *ability to design complex control circuits* to execute an instruction was the central design concern as opposed to *the speed* of decoding or an ALU operation
- Programmer's view of the machine was inseparable from the actual hardware implementation

Accumulator-based computing

- *Single Accumulator*
  - Calculator design carried over to computers

Why?

Registers expensive

The Earliest Instruction Sets

Burks, Goldstein & von Neumann ~1946

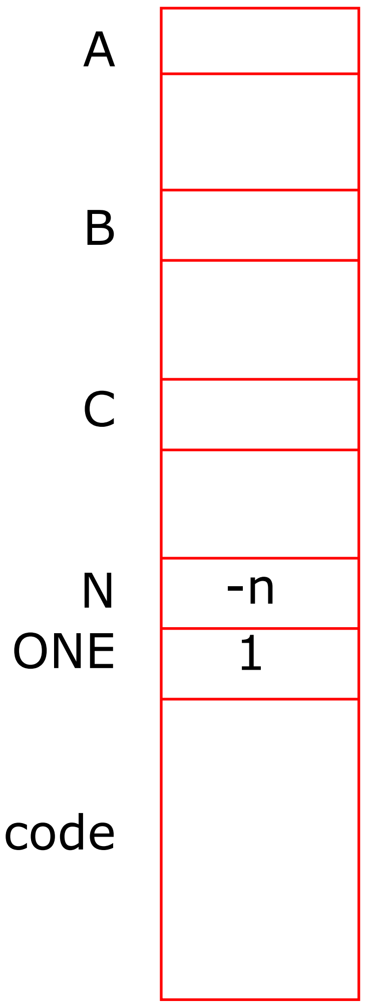
LOAD	x	$AC \leftarrow M[x]$
STORE	x	$M[x] \leftarrow (AC)$
ADD	x	$AC \leftarrow (AC) + M[x]$
SUB	x	
MUL	x	Involved a quotient register
DIV	x	
SHIFT LEFT		$AC \leftarrow 2 \times (AC)$
SHIFT RIGHT		
JUMP	x	$PC \leftarrow x$
JGE	x	if $(AC) \geq 0$ then $PC \leftarrow x$
LOAD ADR	x	$AC \leftarrow$ Extract address field(M[x])
STORE ADR	x	

Typically less than 2 dozen instructions!

Programming: Single Accumulator Machine

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
LOOP   LOAD    N
       JGE     DONE
       ADD     ONE
       STORE   N
F1     LOAD    A
F2     ADD     B
F3     STORE   C
       JUMP    LOOP
DONE   HLT
```



Problem?

How to modify the addresses A, B and C ?

Self-Modifying Code

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
LOOP  LOAD        N
      JGE         DONE
      ADD         ONE
      STORE       N
F1    LOAD        A
F2    ADD         B
F3    STORE       C
      LOAD ADR    F1
      ADD         ONE
      STORE ADR   F1
      LOAD ADR    F2
      ADD         ONE
      STORE ADR   F2
      LOAD ADR    F3
      ADD         ONE
      STORE ADR   F3
      JUMP        LOOP
DONE  HLT
```

modify the program for the next iteration

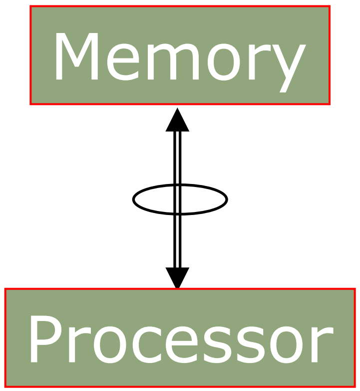
Each iteration involves

	total	*book-keeping*
instruction fetches	*17*	*14*
operand fetches	*10*	*8*
stores	*5*	*4*

Most of the executed instructions are for bookkeeping!

Processor-Memory Bottleneck: Early Solutions

- Indexing capability
  - to reduce bookkeeping instructions
- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator
  - to reduce loads/stores
- Complex instructions
  - to reduce instruction fetches
- Compact instructions
  - implicit address bits for operands
  - to reduce instruction fetch cost



Index Registers
Tom Kilburn, Manchester University, mid 50's

One or more specialized registers to simplify address calculation

Modify existing instructions

LOAD	x, IX	AC ← M[x + (IX)]
ADD	x, IX	AC ← (AC) + M[x + (IX)]
...		

Add new instructions to manipulate *index registers*

JZi	x, IX	if (IX)=0 *then* PC ← x
		else IX ← (IX) + 1
LOADi	x, IX	IX ← M[x] (truncated to fit IX)
...		

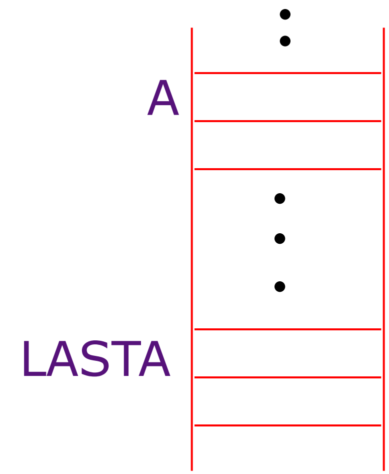
Index registers have accumulator-like characteristics

Using Index Registers

$C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n$

```
        LOADi   N, IX
LOOP    JZi     DONE, IX
        LOAD    LASTA, IX
        ADD     LASTB, IX
        STORE   LASTC, IX
        JUMP    LOOP
DONE    HALT
```

N starts with -n



- *Program does not modify itself*
- *Efficiency has improved dramatically (ops / iter)*

	with index regs	without index regs
instruction fetch	5(2)	17 (14)
operand fetch	2	10 (8)
store	1	5 (4)

- *Costs?*
  - *Complex control*
  - Index register computations (ALU-like circuitry)
  - Instructions 1 to 2 bits longer

Operations on Index Registers

To increment index register by k

$AC \leftarrow (IX)$ *new instruction*
$AC \leftarrow (AC) + k$
$IX \leftarrow (AC)$ *new instruction*

also the AC must be saved and restored

It may be better to increment IX directly

INCi k, IX $IX \leftarrow (IX) + k$

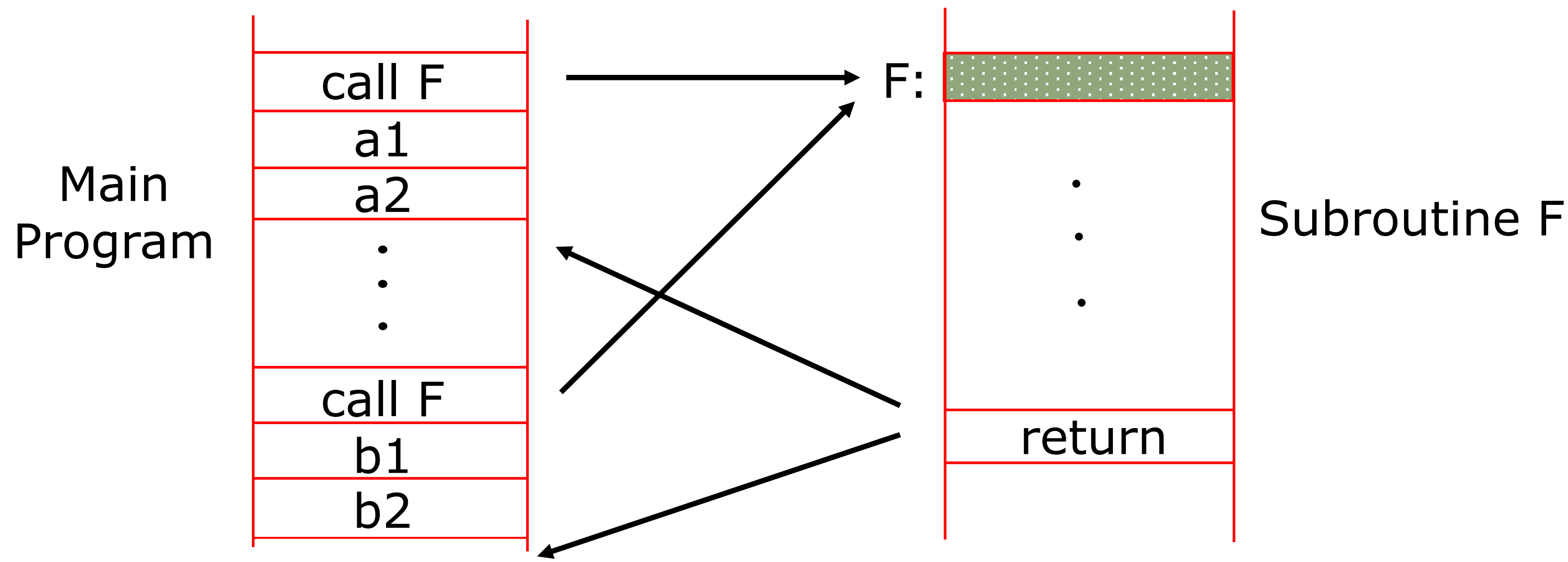
More instructions to manipulate index register

STOREi x, IX $M[x] \leftarrow (IX)$ (extended to fit a word)
...

IX begins to look like an accumulator

$\Rightarrow$ several index registers
several accumulators
$\Rightarrow$ *General Purpose Registers*

Support for Subroutine Calls



A special *subroutine jump instruction*

A: JSR F

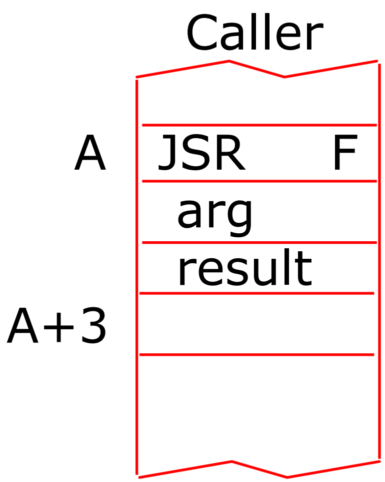
M[F] $\leftarrow$ A + 1 and
jump to F + 1

Indirect Addressing and Subroutine Calls

Indirect addressing

LOAD (x) means AC ← M[M[x]]

...



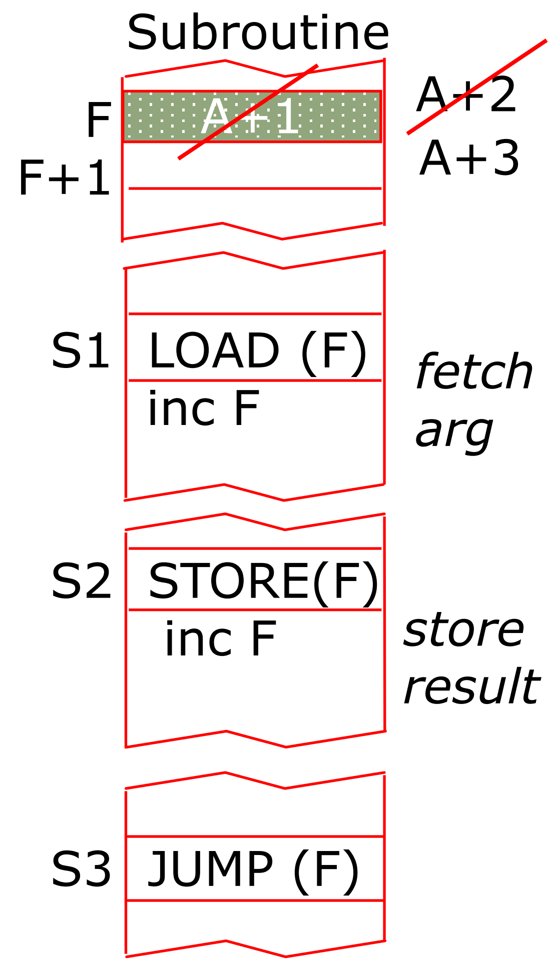
Events:

Execute A

Execute S1

Execute S2

Execute S3



Indirect addressing almost eliminates the need to write self-modifying code (location F still needs to be modified)

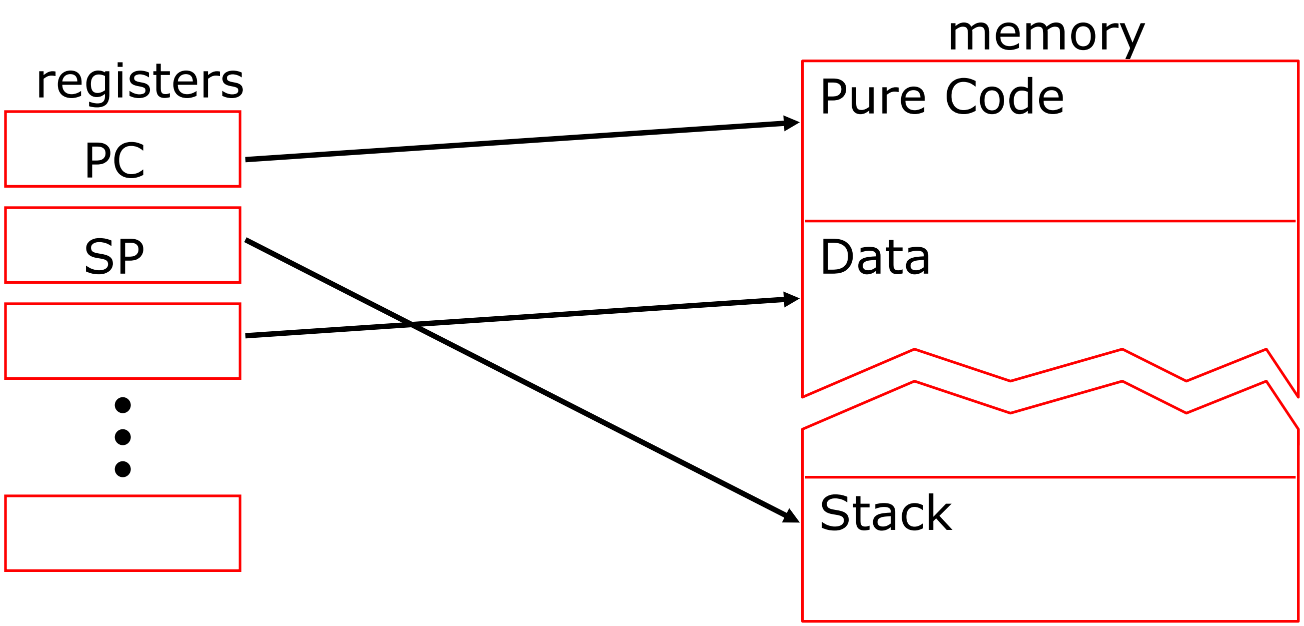
Problems? ⇒ *recursive procedure calls*

Recursive Procedure Calls and Reentrant Codes

Indirect Addressing through a register

LOAD R_1, (R_2)

Load register R_1 with the contents of the word whose address is contained in register R_2



Evolution of Addressing Modes

1. Single accumulator, absolute address

   LOAD x

2. Single accumulator, index registers

   LOAD x, IX

3. Indirection

   LOAD (x)

4. Multiple accumulators, index registers, indirection

   LOAD R, IX, x

   or LOAD R, IX, (x)

   the meaning?

   $R \leftarrow M[M[x] + (IX)]$

   or $R \leftarrow M[M[x + (IX)]]$

5. Indirect through registers

   LOAD R_I, (R_J)

6. The works

   LOAD R_I, R_J, (R_K)

   R_J = index, R_K = base addr

Variety of Instruction Formats

- *Three address formats:* One destination and up to two operand sources per instruction

  (Reg op Reg) to Reg $R_I \leftarrow (R_J)$ op (R_K)
  (Reg op Mem) to Reg $R_I \leftarrow (R_J)$ op $M[x]$

  - x can be specified directly or via a register
  - effective address calculation for x could include indexing, indirection, ...

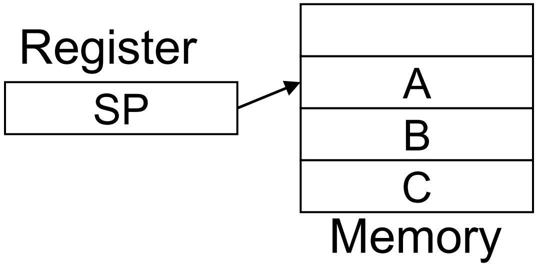
- *Two address formats:* the destination is same as one of the operand sources

  (Reg op Reg) to Reg $R_I \leftarrow (R_I)$ op (R_J)
  (Reg op Mem) to Reg $R_I \leftarrow (R_I)$ op $M[x]$

More Instruction Formats

- *One address formats:* Accumulator machines
  - Accumulator is always other implicit operand

- *Zero address formats:* operands on a stack



add $M[sp-1] \leftarrow M[sp] + M[sp-1]$
load $M[sp] \leftarrow M[M[sp]]$

  - Stack can be in registers or in memory
    - usually top of stack cached in registers

Many different formats are possible!

Instruction sets in the mid 50's

- Great variety of instruction sets, but all intimately tied to implementation details
- Programmer's view of the machine was inseparable from the actual hardware implementation!

Next Lecture:

Instruction Set Architectures: Decoupling Interface and Implementation