6.823 Computer System Architecture

Instructors: Daniel Sanchez and Joel Emer
TA: Hyun Ryong (Ryan) Lee

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?
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
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

- Application
- Algorithm
- Programming Language
- Operating System/Virtual Machine
- Instruction Set Architecture (ISA)
- Microarchitecture
- Register-Transfer Level (RTL)
- Circuits
- Devices
- Physics
Abstraction layers

Original domain of the computer architect ('50s-'80s)

Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Circuits
Devices
Physics
Abstraction layers

Original domain of the computer architect ('50s-'80s)

Domain of computer architecture ('90s)

- Application
- Algorithm
- Programming Language
- Operating System/Virtual Machine
- Instruction Set Architecture (ISA)
- Microarchitecture
- Register-Transfer Level (RTL)
- Circuits
- Devices
- Physics
Abstraction layers

Original domain of the computer architect ('50s-'80s)

- Application
- Algorithm
- Programming Language
- Operating System/Virtual Machine
- Instruction Set Architecture (ISA)
- Microarchitecture
- Register-Transfer Level (RTL)
- Circuits
- Devices
- Physics

Domain of computer architecture ('90s)

Expansion of computer architecture, mid-2000s onward.
Abstraction layers

Original domain of the computer architect ('50s-'80s)

Domain of computer architecture ('90s)

Reliability, power

Expansion of computer architecture, mid-2000s onward.

Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Circuits
Devices
Physics
Abstraction layers

Original domain of the computer architect ('50s-'80s)

Domain of computer architecture ('90s)

Reliability, power

Expansion of computer architecture, mid-2000s onward.

Application
Algorithm
Programming Language
Operating System/Virtual Machine
Instruction Set Architecture (ISA)
Microarchitecture
Register-Transfer Level (RTL)
Circuits
Devices
Physics

Parallel computing, specialization, security, ...

September 8, 2021
Computer Architecture is the design of 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
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:*
    • *Toolchains, operating systems, libraries*
  – Enticement for application innovation
Technology is the dominant factor in computer design
Technology is the dominant factor in computer design

Transistors
Integrated circuits
VLSI (initially)
Flash memories, ...
Technology is the dominant factor in computer design

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

Technology
- Core memories
- Magnetic tapes
- Disks

Computers
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

Computers
But Software...
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*
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.
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:
Architecture is engineering design under constraints

Factors to consider:
  • Performance of whole system on target applications
    – Average case & worst case
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
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
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
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
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
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 on Monday and Wednesday
  – 1:00pm to 2:30pm in room 32-141

• Tutorial on Friday
  – 1:00pm to 2:00pm in room 32-141
  – 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 32-141
  – Attendance is NOT optional

• Instructor office hours
  – After class or by email appointment

• TA office hours
  – Thursday 4-5:30pm @ Stata 32G-725
“New normal” policies

• We’re excited to return to the classroom, but want everyone to be and feel safe

• We’ll record videos of lectures and tutorials for students who need to miss lecture
  – Due to isolation/quarantine, visa issues, case spikes, etc.
  – However, these videos will be best-effort and more basic than for online classes (e.g., no webcam feed, audio may be worse)
  – Please do not use these to take 6.823 as an online course

• When asking questions, please keep your mask on

• If you feel uncomfortable with any aspect of our in-person interactions, please let us know
Online resources & help

• We use Piazza extensively
  – Fastest way to get your questions answered
  – All course announcements are made on Piazza

• This is not a normal term; if you need help, let us know!
  – We can be accommodating
The course has three modules

**Module 1**
- ISA and Simple In-Order Pipelines
- Caches and Virtual Memory
- Complex Pipelining and Out-of-Order Execution
- Branch Prediction and Speculative Execution

**Module 2**
- Multithreading and Multiprocessors
- Coherence and consistency
- On-chip networks

**Module 3**
- Microcoding and VLIW
- Vector machines and GPUs
- Hardware accelerators
- Hardware security
The course has three modules

Module 1
- ISA and Simple In-Order Pipelines
- Caches and Virtual Memory
- Complex Pipelining and Out-of-Order Execution
- Branch Prediction and Speculative Execution

Module 2
- Multithreading and Multiprocessors
- Coherence and consistency
- On-chip networks

Module 3
- Microcoding and VLIW
- Vector machines and GPUs
- Hardware accelerators
- Hardware security

New this term
Textbook and readings

  - 5th edition 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 three closed book 1.5 hour quizzes
  - The first two quizzes will be held during the tutorials; the last one during the last lecture (dates on web syllabus)
  - We’ll have makeups if needed
- 25% of the grade is based on four laboratory exercises
- No final exam
- No final project

September 8, 2021
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
  – Based on widely-used PIN tool
  – Labs 2 and 4 are open-ended challenges

• You must complete labs & quizzes individually
  – Please review the collaboration & academic honesty policy
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 by Friday (on recitation or send answers to TA mailing list)

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!
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!

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”
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”
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*
    \[ \Rightarrow \text{“program can be manipulated as data”} \]
- *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
Stored Program Computer

Program = A sequence of instructions

*How to control instruction sequencing?*
Program = A sequence of instructions

How to control instruction sequencing?
manual control
calculators
Program = A sequence of instructions

How to control instruction sequencing?

manual control

automatic control
  external (paper tape)

calculators

Harvard Mark I, 1944
Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?

manual control

automatic control
external (paper tape)

internal
plug board
read-only memory
read-write memory

Harvard Mark I, 1944
Zuse’s Z1, WW2
ENIAC 1946
ENIAC 1948
EDVAC 1947 (concept)

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

Program = A sequence of instructions

How to control instruction sequencing?

manual control

automatic control
  external (paper tape)
  internal
    plug board
    read-only memory
    read-write memory

Harvard Mark I, 1944
Zuse’s Z1, WW2
ENIAC 1946
ENIAC 1948
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

<table>
<thead>
<tr>
<th>Institution</th>
<th>Location</th>
<th>Year</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>Princeton</td>
<td>46-52</td>
<td>Bigelow</td>
</tr>
<tr>
<td>EDSAC</td>
<td>Cambridge</td>
<td>46-50</td>
<td>Wilkes</td>
</tr>
<tr>
<td>MANIAC</td>
<td>Los Alamos</td>
<td>49-52</td>
<td>Metropolis</td>
</tr>
<tr>
<td>JOHNIAC</td>
<td>Rand</td>
<td>50-53</td>
<td></td>
</tr>
<tr>
<td>ILLIAC</td>
<td>Illinois</td>
<td>49-52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Argonne</td>
<td>49-53</td>
<td></td>
</tr>
<tr>
<td>SWAC</td>
<td>UCLA-NBS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Spread of Ideas

ENIAC & EDVAC had immediate impact

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

<table>
<thead>
<tr>
<th>Computer</th>
<th>Institution</th>
<th>Years</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>Princeton</td>
<td>46-52</td>
<td>Bigelow</td>
</tr>
<tr>
<td>EDSAC</td>
<td>Cambridge</td>
<td>46-50</td>
<td>Wilkes</td>
</tr>
<tr>
<td>MANIAC</td>
<td>Los Alamos</td>
<td>49-52</td>
<td>Metropolis</td>
</tr>
<tr>
<td>JOHNIAC</td>
<td>Rand</td>
<td>50-53</td>
<td></td>
</tr>
<tr>
<td>ILLIAC</td>
<td>Illinois</td>
<td>49-52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Argonne</td>
<td>49-53</td>
<td></td>
</tr>
<tr>
<td>SWAC</td>
<td>UCLA-NBS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

UNIVAC - the first commercial computer, 1951
The Spread of Ideas

**ENIAC & EDVAC had immediate impact**

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

<table>
<thead>
<tr>
<th>Computer</th>
<th>Institution</th>
<th>Years</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>Princeton</td>
<td>46-52</td>
<td>Bigelow</td>
</tr>
<tr>
<td>EDSAC</td>
<td>Cambridge</td>
<td>46-50</td>
<td>Wilkes</td>
</tr>
<tr>
<td>MANIAC</td>
<td>Los Alamos</td>
<td>49-52</td>
<td>Metropolis</td>
</tr>
<tr>
<td>JOHNIAC</td>
<td>Rand</td>
<td>50-53</td>
<td></td>
</tr>
<tr>
<td>ILLIAC</td>
<td>Illinois</td>
<td>49-52</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Argonne</td>
<td>49-53</td>
<td></td>
</tr>
<tr>
<td>SWAC</td>
<td>UCLA-NBS</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**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
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
Computers in the mid 50’s

• Hardware was expensive
Computers in the mid 50’s

• Hardware was expensive
• Stores were small (1000 words)
Computers in the mid 50’s

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

⇒ No resident system-software!
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
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
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
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
Accumulator-based computing

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

Why?
Accumulator-based computing

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

**Why?**

*Registers expensive*
The Earliest Instruction Sets

*Burks, Goldstein & von Neumann*  ~1946
## The Earliest Instruction Sets

**Burks, Goldstein & von Neumann ~1946**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
</tr>
</tbody>
</table>
# The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALU Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD x</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE x</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD x</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB x</td>
<td></td>
</tr>
<tr>
<td>MUL x</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV x</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
</tr>
<tr>
<td>JUMP x</td>
<td>PC ← x</td>
</tr>
<tr>
<td>JGE x</td>
<td>if (AC) ≥ 0 then PC ← x</td>
</tr>
</tbody>
</table>
### The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>x</td>
<td>AC ← M[x]</td>
</tr>
<tr>
<td>STORE</td>
<td>x</td>
<td>M[x] ← (AC)</td>
</tr>
<tr>
<td>ADD</td>
<td>x</td>
<td>AC ← (AC) + M[x]</td>
</tr>
<tr>
<td>SUB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL</td>
<td>x</td>
<td>Involved a quotient register</td>
</tr>
<tr>
<td>DIV</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td></td>
<td>AC ← 2 × (AC)</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>x</td>
<td>PC ← x</td>
</tr>
<tr>
<td>JGE</td>
<td>x</td>
<td>if (AC) ≥ 0 then PC ← x</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>x</td>
<td>AC ← Extract address field(M[x])</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>x</td>
<td></td>
</tr>
</tbody>
</table>
# The Earliest Instruction Sets

*Burks, Goldstein & von Neumann ~1946*

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>$x \rightarrow AC \leftarrow M[x]$</td>
</tr>
<tr>
<td>STORE</td>
<td>$x \rightarrow M[x] \leftarrow (AC)$</td>
</tr>
<tr>
<td>ADD</td>
<td>$x \rightarrow AC \leftarrow (AC) + M[x]$</td>
</tr>
<tr>
<td>SUB</td>
<td>$x \rightarrow AC \leftarrow (AC) + M[x]$</td>
</tr>
<tr>
<td>MUL</td>
<td>$x \rightarrow AC \leftarrow 2 \times (AC)$</td>
</tr>
<tr>
<td>DIV</td>
<td>$x \rightarrow AC \leftarrow 2 \times (AC)$</td>
</tr>
<tr>
<td>JUMP</td>
<td>$x \rightarrow PC \leftarrow x$</td>
</tr>
<tr>
<td>JGE</td>
<td>$x \rightarrow PC \leftarrow x$</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>$x \rightarrow AC \leftarrow $ Extract address field(M[x])</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>$x \rightarrow AC \leftarrow $ Extract address field(M[x])</td>
</tr>
</tbody>
</table>

*Typically less than 2 dozen instructions!*
Programming: Single Accumulator Machine

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
Programming: Single Accumulator Machine

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>A</td>
</tr>
<tr>
<td>LOAD</td>
<td>B</td>
</tr>
<tr>
<td>JGE</td>
<td>C</td>
</tr>
<tr>
<td>ADD</td>
<td>N</td>
</tr>
<tr>
<td>STORE</td>
<td>ONE</td>
</tr>
<tr>
<td>F1 LOAD</td>
<td>N</td>
</tr>
<tr>
<td>F2 ADD</td>
<td>ONE</td>
</tr>
<tr>
<td>F3 STORE</td>
<td>N</td>
</tr>
<tr>
<td>JUMP</td>
<td>ONE</td>
</tr>
<tr>
<td>DONE HLT</td>
<td>C</td>
</tr>
</tbody>
</table>

Code: \(-n, 1\)
Programming: Single Accumulator Machine

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOAD N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JGE DONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD ONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE N</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1 LOAD A</td>
<td></td>
<td>-n</td>
<td>1</td>
</tr>
<tr>
<td>F2 ADD B</td>
<td></td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>F3 STORE C</td>
<td></td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>JUMP LOOP</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DONE HLT</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Problem?
Programming: Single Accumulator Machine

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

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td>ONE</td>
</tr>
<tr>
<td>STORE</td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
<tr>
<td>JUMP</td>
<td></td>
<td>LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>

Problem?

How to modify the addresses A, B and C?
Self-Modifying Code

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
# Self-Modifying Code

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

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>N</td>
<td></td>
</tr>
</tbody>
</table>

| F1   | LOAD | A |
| F2   | ADD  | B |
| F3   | STORE| C |

| DONE | HLT  | LOOP |

modify the program for the next iteration
Self-Modifying Code

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JGE</td>
<td>DONE</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>N</td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
</tbody>
</table>

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

modify the program for the next iteration

<table>
<thead>
<tr>
<th>LOAD ADR</th>
<th>F1</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F1</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>F2</td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F2</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>F3</td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JUMP</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>DONE</td>
<td>HLT</td>
</tr>
</tbody>
</table>
## Self-Modifying Code

Each iteration involves the following operations:

- **LOAD** instruction fetches
- **ADD** operand fetches
- **STORE** stores

The target equation is:

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

### Code Snippet

```
LOOP
  LOAD N
  JGE DONE
  ADD ONE
  STORE N

F1 LOAD A
F2 ADD B
F3 STORE C
JUMP LOOP
DONE HLT

modify the program for the next iteration

modify the
program
for the next
iteration

```
Self-Modifying Code

Each iteration involves total book-keeping:

- Instruction fetches: 17
- Operand fetches
- Stores

\[ C_i \leftarrow A_i + B_i, \quad 1 \leq i \leq n \]
Self-Modifying Code

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td>JGE</td>
<td>DONE</td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td>ONE</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>N</td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td>F1</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td>F3</td>
</tr>
<tr>
<td></td>
<td>JUMP</td>
<td>LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>

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

Each iteration involves:

- total book-keeping: 17
- instruction fetches: 17
- operand fetches: 10
- stores

modify the program for the next iteration

September 8, 2021
Self-Modifying Code

<table>
<thead>
<tr>
<th>Instruction</th>
<th>LOOPS</th>
<th>LOAD</th>
<th>N</th>
<th>JUMP</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JGE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADD</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>LOAD ADR</td>
<td>F1</td>
<td>ONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>LOAD ADR</td>
<td>F2</td>
<td>ONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>LOAD ADR</td>
<td>F3</td>
<td>ONE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>ADD</td>
<td>ONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>ONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>ADD</td>
<td>ONE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F1</td>
<td>STORE ADR</td>
<td>F1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F2</td>
<td>STORE ADR</td>
<td>F2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>F3</td>
<td>STORE ADR</td>
<td>F3</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

Each iteration involves

- **total book-keeping**: 17
- **instruction fetches**: 17
- **operand fetches**: 10
- **stores**: 5

*modify the program for the next iteration*
Self-Modifying Code

Each iteration involves

- total book-keeping:
  - instruction fetches: 17 + 14
  - operand fetches: 10
  - stores: 5

C_i ← A_i + B_i, 1 ≤ i ≤ n

modify the program for the next iteration

<table>
<thead>
<tr>
<th>LOOP</th>
<th>LOAD</th>
<th>N</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>JGE</td>
<td>DONE</td>
</tr>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>N</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1</th>
<th>LOAD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td></td>
<td>STORE</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F2</th>
<th>LOAD ADR</th>
<th>ONE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADD</td>
<td>F2</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td>F2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F3</th>
<th>LOAD ADR</th>
<th>F3</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ADD</td>
<td>ONE</td>
</tr>
<tr>
<td></td>
<td>STORE ADR</td>
<td>F3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>DONE</th>
<th>JUMP</th>
<th>LOOP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>
Self-Modifying Code

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

Each iteration involves
- instruction fetches: 17
- operand fetches: 10
- stores: 5

modify the program for the next iteration

<table>
<thead>
<tr>
<th>Instruction</th>
<th>N</th>
<th>STORE</th>
<th>LOAD</th>
<th>ADD</th>
<th>JUMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>LOAD</td>
<td>N</td>
<td>STORE</td>
<td>LOAD</td>
<td>ADD</td>
</tr>
<tr>
<td>F1</td>
<td>LOAD</td>
<td>A</td>
<td>ADD</td>
<td>F1</td>
<td>STORE</td>
</tr>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
<td>F2</td>
<td>STORE</td>
<td>LOAD</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
<td>F3</td>
<td>LOAD</td>
<td>ADD</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR</td>
<td>F1</td>
<td>ADD</td>
<td>STORE ADR</td>
<td>STORE ADR</td>
</tr>
<tr>
<td></td>
<td>JUMP ADR</td>
<td>F2</td>
<td>ADD</td>
<td>STORE ADR</td>
<td>STORE ADR</td>
</tr>
<tr>
<td></td>
<td>HLT</td>
<td>LOOP</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Self-Modifying Code

C_i ← A_i + B_i, \ 1 \leq i \leq n

Each iteration involves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Total Bookkeeping</th>
<th>Total Fetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instructions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fetches</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>Stores</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

modify the program for the next iteration
Self-Modifying Code

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

Each iteration involves

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Total bookkeeping</th>
<th>Bookkeeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction fetches</td>
<td>17</td>
<td>14</td>
</tr>
<tr>
<td>Operand fetches</td>
<td>10</td>
<td>8</td>
</tr>
<tr>
<td>Stores</td>
<td>5</td>
<td>4</td>
</tr>
</tbody>
</table>

Most of the executed instructions are for bookkeeping!
Processor-Memory Bottleneck: Early Solutions

- Indexing capability

- Fast local storage in the processor
  - 8-16 registers as opposed to one accumulator

- Complex instructions

- Compact instructions
  - implicit address bits for operands
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

- Complex instructions

- Compact instructions
  - implicit address bits for operands
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**

- **Compact instructions**
  - implicit address bits for operands
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
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
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)]
...
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)
...

September 8, 2021

MIT 6.823 Fall 2021
Index Registers
Tom Kilburn, Manchester University, mid 50’s

One or more specialized registers to simplify address calculation

Modify existing instructions
- LOAD x, IX
  \[ AC \leftarrow M[x + (IX)] \]
- ADD x, IX
  \[ AC \leftarrow (AC) + M[x + (IX)] \]
...

Add new instructions to manipulate index registers
- JZi x, IX
  if (IX)=0 then \[ PC \leftarrow x \]
  else \[ IX \leftarrow (IX) + 1 \]
- LOADi x, IX
  \[ IX \leftarrow M[x] \] (truncated to fit IX)
...

Index registers have accumulator-like characteristics
Using Index Registers

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

<table>
<thead>
<tr>
<th></th>
<th>LOAD</th>
<th>ADD</th>
<th>STORE</th>
<th>JUMP</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>\text{LOADi} N, IX</td>
<td>\text{ADD} \text{LASTB, IX}</td>
<td>\text{STORE} \text{LASTC, IX}</td>
<td>\text{JUMP} \text{LOOP}</td>
<td>\text{DONE} HALT</td>
</tr>
</tbody>
</table>

\( N \) starts with \(-n\)

\[ A \]

\[ \text{LASTA} \]
Using Index Registers

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

- Program does not modify itself

<table>
<thead>
<tr>
<th>Loop</th>
<th>Instruction</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Load</td>
<td>LOADi N, IX</td>
<td>LASTA, IX</td>
</tr>
<tr>
<td>Add</td>
<td>ADD LASTB, IX</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>STORE LASTC, IX</td>
<td></td>
</tr>
<tr>
<td>Jump</td>
<td>JUMP LOOP</td>
<td></td>
</tr>
</tbody>
</table>

\[ N \text{ starts with } -n \]

A

\[
\begin{array}{c}
A \\
\vdots \\
A \\
\vdots \\
A
\end{array}
\]

LASTA

\[
\begin{array}{c}
\text{LASTA} \\
\vdots \\
\text{LASTA} \\
\vdots \\
\text{LASTA}
\end{array}
\]
# Using Index Registers

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Source</th>
<th>Destination</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD $i$</td>
<td>N, IX</td>
<td>IX</td>
</tr>
<tr>
<td>LOOP</td>
<td></td>
<td>N starts with -n</td>
</tr>
<tr>
<td>JZ$i$</td>
<td>DONE, IX</td>
<td></td>
</tr>
<tr>
<td>LOAD</td>
<td>LASTA, IX</td>
<td>IX</td>
</tr>
<tr>
<td>ADD</td>
<td>LASTB, IX</td>
<td></td>
</tr>
<tr>
<td>STORE</td>
<td>LASTC, IX</td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>HALT</td>
<td></td>
</tr>
</tbody>
</table>

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

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

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter) with index regs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>With Index Regs</th>
<th>Without Index Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>17 (14)</td>
<td></td>
</tr>
<tr>
<td>Operand Fetch</td>
<td>10 (8)</td>
<td></td>
</tr>
<tr>
<td>Store</td>
<td>5 (4)</td>
<td></td>
</tr>
</tbody>
</table>

N starts with -n
Using Index Registers

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

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

<table>
<thead>
<tr>
<th>Operation</th>
<th>With Index Regs</th>
<th>Without Index Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>Operand Fetch</td>
<td></td>
<td>10 (8)</td>
</tr>
<tr>
<td>Store</td>
<td></td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

\[ N \text{ starts with } -n \]
Using Index Registers

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

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter) with index regs

<table>
<thead>
<tr>
<th>Instruction</th>
<th>With Index Regs</th>
<th>Without Index Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction Fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>Operand Fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>Store</td>
<td>5 (4)</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

N starts with \(-n\)
Using Index Registers

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

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

<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>1</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

\[ A_i \leftarrow \text{LASTA}, \quad \text{LASTB}, \quad \text{LASTC}, \quad \text{LOOP} \]

\[ \text{LOAD}_i \quad \text{N, IX} \]

\[ \text{ADD}_i \quad \text{LASTB, IX} \]

\[ \text{STORE}_i \quad \text{LASTC, IX} \]

\[ \text{JUMP}_i \quad \text{LOOP} \]

\[ \text{DONE}_i \quad \text{HALT} \]

\[ \text{N starts with } -n \]
Using Index Registers

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

- Program does not modify itself
- Efficiency has improved dramatically (ops / iter) with index regs

<table>
<thead>
<tr>
<th></th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetch</td>
<td>5(2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>operand fetch</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td>1</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>

- Costs?
Using Index Registers

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

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Function</th>
<th>with index regs</th>
<th>without index regs</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>N, IX</td>
<td>5 (2)</td>
<td>17 (14)</td>
</tr>
<tr>
<td>ADD</td>
<td>LASTB, IX</td>
<td>2</td>
<td>10 (8)</td>
</tr>
<tr>
<td>STORE</td>
<td>LASTC, IX</td>
<td>1</td>
<td>5 (4)</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

• **Program does not modify itself**

• **Efficiency has improved dramatically (ops / iter)**

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

To increment index register by k

\[
\begin{align*}
& AC \leftarrow (IX) \quad \text{new instruction} \\
& AC \leftarrow (AC) + k \\
& IX \leftarrow (AC) \quad \text{new instruction}
\end{align*}
\]
Operations on Index Registers

To increment index register by \( k \)

\[
\begin{align*}
\text{AC} & \leftarrow (\text{IX}) & \text{new instruction} \\
\text{AC} & \leftarrow (\text{AC}) + k \\
\text{IX} & \leftarrow (\text{AC}) & \text{new instruction}
\end{align*}
\]

also the AC must be saved and restored
Operations on Index Registers

To increment index register by $k$

\[
\begin{align*}
    AC &\leftarrow (IX) \quad \text{new instruction} \\
    AC &\leftarrow (AC) + k \\
    IX &\leftarrow (AC) \quad \text{new instruction}
\end{align*}
\]

also the AC must be saved and restored

It may be better to increment IX directly

\[
\begin{align*}
    \text{INCi} &\quad k, \, IX \quad IX \leftarrow (IX) + k
\end{align*}
\]
Operations on Index Registers

To increment index register by \( k \)

\[
\begin{align*}
AC & \leftarrow (IX) \\
AC & \leftarrow (AC) + k \\
IX & \leftarrow (AC)
\end{align*}
\]

It may be better to increment IX directly

\[
\begin{align*}
\text{INCI} & \quad k, IX \quad IX \leftarrow (IX) + k
\end{align*}
\]

More instructions to manipulate index register

\[
\begin{align*}
\text{STOREI} & \quad x, IX \quad M[x] \leftarrow (IX) \text{ (extended to fit a word)}
\end{align*}
\]

...
Operations on Index Registers

To increment index register by k

\[ AC \leftarrow (IX) \quad \text{new instruction} \]
\[ AC \leftarrow (AC) + k \]
\[ IX \leftarrow (AC) \quad \text{new instruction} \]

also the AC must be saved and restored

It may be better to increment IX directly

\[ \text{INCI} \quad k, \text{IX} \quad IX \leftarrow (IX) + k \]

More instructions to manipulate index register

\[ \text{STOREi} \quad x, \text{IX} \quad M[x] \leftarrow (IX) \quad \text{(extended to fit a word)} \]

... 

IX begins to look like an accumulator

⇒ several index registers

⇒ several accumulators

⇒ General Purpose Registers
Evolution of Addressing Modes
Evolution of Addressing Modes

1. Single accumulator, absolute address
   
   LOAD x
Evolution of Addressing Modes

1. Single accumulator, absolute address
   LOAD x

2. Single accumulator, index registers
   LOAD x, IX
Evolution of Addressing Modes

1. Single accumulator, absolute address
   LOAD  x

2. Single accumulator, index registers
   LOAD  x, IX

3. Indirection
   LOAD  (x)
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 ← M[M[x] + (IX)]
   or R ← M[M[x + (IX)]]
Evolution of Addressing Modes

1. Single accumulator, absolute address
   \[\text{LOAD } x\]

2. Single accumulator, index registers
   \[\text{LOAD } x, \text{IX}\]

3. Indirection
   \[\text{LOAD } (x)\]

4. Multiple accumulators, index registers, indirection
   \[\text{LOAD } R, \text{IX, } x\]
   \[\text{or } \text{LOAD } R, \text{IX}, (x)\]
   the meaning?
   \[R \leftarrow M[M[x] + (\text{IX})]\]
   \[\text{or } R \leftarrow M[M[x + (\text{IX})]]\]

5. Indirect through registers
   \[\text{LOAD } R_I, (R_J)\]
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
Variety of Instruction Formats

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

  \[
  \begin{align*}
  (\text{Reg op Reg}) \text{ to Reg} & \quad R_i \leftarrow (R_j) \text{ op } (R_k) \\
  (\text{Reg op Mem}) \text{ to Reg} & \quad R_i \leftarrow (R_j) \text{ op } M[x]
  \end{align*}
  \]

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

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

  \[
  \begin{align*}
  \text{(Reg op Reg) to Reg} & \quad R_i \leftarrow (R_j) \text{ op } (R_k) \\
  \text{(Reg op Mem) to Reg} & \quad R_i \leftarrow (R_j) \text{ op } M[x]
  \end{align*}
  \]

  – 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

  \[
  \begin{align*}
  \text{(Reg op Reg) to Reg} & \quad R_i \leftarrow (R_i) \text{ op } (R_j) \\
  \text{(Reg op Mem) to Reg} & \quad R_i \leftarrow (R_i) \text{ op } M[x]
  \end{align*}
  \]
More Instruction Formats
More Instruction Formats

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

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

• **Zero address formats:** operands on a stack

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

```
load M[sp] ← M[M[sp]]
```
More Instruction Formats

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

- **Zero address formats**: operands on a stack
  - Stack can be in registers or in memory
    - usually top of stack cached in registers

```
load   M[sp] ← M[M[sp]]
```

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!
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