6.823 Computer System Architecture

Lecturer: Daniel Sanchez
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

Application

Algorithm

Programming Language

Operating System/Virtual Machine

Instruction Set Architecture (ISA)

Microarchitecture

Register-Transfer Level (RTL)

Circuits

Devices

Physics

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

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

Domain of computer architecture ('90s)
Abstraction layers

Original domain of the computer architect (‘50s–‘80s)

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.
Abstraction layers

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

Parallel computing security, ...

Domain of computer architecture ('90s)

Reliability, power

Expansion of computer architecture, mid-2000s onward.
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

Technology

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

Computers
Technology is the dominant factor in computer design

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

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

Computers

February 6, 2019
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*
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.
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 Mondays and Wednesdays
  – 1:00pm to 2:30pm in room 6-120

• Tutorial on Fridays
  – 1:00pm to 2:00pm in room 32-155
  – 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 32-155
  – 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

  – 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 project
  - 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 $\mu$s, 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*
    ⇒ “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?
Stored Program Computer

Program = A sequence of instructions

How to control instruction sequencing?
manual control  calculators
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
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

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

EDSAC 1950 Maurice Wilkes

Harvard Mark I, 1944
Zuse’s Z1, WW2
ENIAC 1946
ENIAC 1948
EDVAC 1947 (concept)
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>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>IAS</td>
<td>Princeton</td>
<td>46-52</td>
</tr>
<tr>
<td>EDSAC</td>
<td>Cambridge</td>
<td>46-50</td>
</tr>
<tr>
<td>MANIAC</td>
<td>Los Alamos</td>
<td>49-52</td>
</tr>
<tr>
<td>JOHNIAC</td>
<td>Rand</td>
<td>50-53</td>
</tr>
<tr>
<td>ILLIAC</td>
<td>Illinois</td>
<td>49-52</td>
</tr>
<tr>
<td>Argonne</td>
<td></td>
<td>49-53</td>
</tr>
<tr>
<td>SWAC</td>
<td>UCLA-NBS</td>
<td></td>
</tr>
</tbody>
</table>

February 6, 2019
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>Location</th>
<th>Years</th>
<th>Research Organization</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**

<table>
<thead>
<tr>
<th>ENIAC</th>
<th>⇒</th>
<th>EDVAC</th>
</tr>
</thead>
<tbody>
<tr>
<td>18,000 tubes</td>
<td>4,000 tubes</td>
<td>2000 word storage</td>
</tr>
<tr>
<td>20 10-digit numbers</td>
<td>2000 word storage</td>
<td>mercury delay lines</td>
</tr>
</tbody>
</table>

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

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

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

Mean time between failures (MTBF)
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>Operation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD x</td>
<td>AC ← M[x]</td>
<td></td>
</tr>
<tr>
<td>STORE x</td>
<td>M[x] ← (AC)</td>
<td></td>
</tr>
<tr>
<td>ADD x</td>
<td>AC ← (AC) + M[x]</td>
<td></td>
</tr>
<tr>
<td>SUB x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MUL x</td>
<td>Involved a quotient register</td>
<td></td>
</tr>
<tr>
<td>DIV x</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SHIFT LEFT</td>
<td>AC ← 2 × (AC)</td>
<td></td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The Earliest Instruction Sets
Burks, Goldstein & von Neumann  ~1946

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Format</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
<td>x</td>
<td>( AC \leftarrow M[x] )</td>
</tr>
<tr>
<td>STORE</td>
<td>x</td>
<td>( M[x] \leftarrow (AC) )</td>
</tr>
<tr>
<td>ADD</td>
<td>x</td>
<td>( AC \leftarrow (AC) + M[x] )</td>
</tr>
<tr>
<td>SUB</td>
<td>x</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>x</td>
<td></td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
<td>x</td>
<td></td>
</tr>
<tr>
<td>JUMP</td>
<td>x</td>
<td>( PC \leftarrow x )</td>
</tr>
<tr>
<td>JGE</td>
<td>x</td>
<td>if ( (AC) \geq 0 ) then ( PC \leftarrow x )</td>
</tr>
</tbody>
</table>
# The Earliest Instruction Sets

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Alpha</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>x</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>
</tr>
</thead>
<tbody>
<tr>
<td>LOAD</td>
</tr>
<tr>
<td>STORE</td>
</tr>
<tr>
<td>ADD</td>
</tr>
<tr>
<td>SUB</td>
</tr>
<tr>
<td>MUL</td>
</tr>
<tr>
<td>DIV</td>
</tr>
<tr>
<td>SHIFT LEFT</td>
</tr>
<tr>
<td>SHIFT RIGHT</td>
</tr>
<tr>
<td>JUMP</td>
</tr>
<tr>
<td>JGE</td>
</tr>
<tr>
<td>LOAD ADR</td>
</tr>
<tr>
<td>STORE ADR</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

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

<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>
<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>LOOP</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td>HLT</td>
<td></td>
</tr>
</tbody>
</table>

code

A
B
C

N

- n

ONE

1
Programming: Single Accumulator Machine

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

**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

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

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

The program below modifies itself for the next iteration. The operation is defined as:

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

```plaintext
<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>
</tbody>
</table>

### Modify the Program for the Next Iteration

- 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
```
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>

modify the program for the next iteration

Each iteration involves

- total book-keeping
- instruction fetches
- operand fetches
- stores

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

Each iteration involves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetches</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>total bookkeeping</strong></td>
<td>17</td>
</tr>
<tr>
<td><strong>instruction fetches</strong></td>
<td></td>
</tr>
<tr>
<td><strong>operand fetches</strong></td>
<td></td>
</tr>
<tr>
<td><strong>stores</strong></td>
<td></td>
</tr>
</tbody>
</table>

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

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

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

modify the program for the next iteration
Self-Modifying Code

Each iteration involves

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Fetches</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instruction</td>
<td>total</td>
</tr>
<tr>
<td></td>
<td>book-keeping</td>
</tr>
<tr>
<td>Fetches</td>
<td></td>
</tr>
<tr>
<td>stores</td>
<td></td>
</tr>
</tbody>
</table>

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

modify the program for the next iteration

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

Each iteration involves:

- Instruction fetches: \(17\) (\(14\) for operand fetches)
- Stores: \(5\)

The program for the next iteration modifies:

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

Each iteration involves

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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Memory Location</th>
<th>Memory Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td>LOAD</td>
<td>N</td>
</tr>
<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>
<tr>
<td>LOAD ADR</td>
<td>F1</td>
<td>ONE</td>
</tr>
<tr>
<td>ADD</td>
<td>F1</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F1</td>
<td>ONE</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>F2</td>
<td>ONE</td>
</tr>
<tr>
<td>ADD</td>
<td>F2</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F2</td>
<td>ONE</td>
</tr>
<tr>
<td>LOAD ADR</td>
<td>F3</td>
<td>ONE</td>
</tr>
<tr>
<td>ADD</td>
<td>F3</td>
<td>ONE</td>
</tr>
<tr>
<td>STORE ADR</td>
<td>F3</td>
<td>ONE</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>HLT</td>
<td>DONE</td>
<td></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>total book-keeping</th>
</tr>
</thead>
<tbody>
<tr>
<td>instruction fetches</td>
</tr>
<tr>
<td>operand fetches</td>
</tr>
<tr>
<td>stores</td>
</tr>
</tbody>
</table>

**modify the program for the next iteration**

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

F1
LOAD A
ADD B
STORE C

F2
LOAD ADR F1
ADD ONE
STORE ADR F1
LOAD ADR F2
ADD ONE
STORE ADR F2
LOAD ADR F3
ADD ONE
STORE ADR F3

F3
JUMP LOOP
DONE
HLT
```
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>
</tbody>
</table>

<table>
<thead>
<tr>
<th>F1</th>
<th>LOAD</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>F2</td>
<td>ADD</td>
<td>B</td>
</tr>
<tr>
<td>F3</td>
<td>STORE</td>
<td>C</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Modifying Code</th>
<th>C&lt;sub&gt;i&lt;/sub&gt; ← A&lt;sub&gt;i&lt;/sub&gt; + B&lt;sub&gt;i&lt;/sub&gt;, 1 ≤ i ≤ n</th>
</tr>
</thead>
<tbody>
<tr>
<td>----------------</td>
<td>-------------------------------------------------</td>
</tr>
<tr>
<td>LOOP</td>
<td>LOAD ADR F1</td>
</tr>
<tr>
<td></td>
<td>ADD ONE F1</td>
</tr>
<tr>
<td></td>
<td>STORE ADR F1</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR F2</td>
</tr>
<tr>
<td></td>
<td>ADD ONE F2</td>
</tr>
<tr>
<td></td>
<td>STORE ADR F2</td>
</tr>
<tr>
<td></td>
<td>LOAD ADR F3</td>
</tr>
<tr>
<td></td>
<td>ADD ONE F3</td>
</tr>
<tr>
<td></td>
<td>STORE ADR F3</td>
</tr>
<tr>
<td></td>
<td>JUMP LOOP</td>
</tr>
</tbody>
</table>

Each iteration involves

<table>
<thead>
<tr>
<th>bookkeeping</th>
<th>total</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
ADD x, IX

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

Add new instructions to manipulate index registers

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

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  

A  
\[ \vdots \]  
\[ \vdots \]  
\[ \vdots \]  

N starts with \(-n\)
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

- Program does not modify itself

N starts with -n

\[ A_i \rightleftharpoons A_{i-1} + B_{i-1}, \quad 1 \leq i \leq n \]

LASTA

\[ C_{n-1} \leftarrow A_{n-1} + B_{n-1} \]
Using Index Registers

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

- **LOADi** \[ N, \text{IX} \]
- **LOAD** \[ \text{LASTA, IX} \]
- **ADD** \[ \text{LASTB, IX} \]
- **STORE** \[ \text{LASTC, IX} \]
- **JUMP** \[ \text{LOOP} \]
- **LOOP**
- **JZi** \[ \text{DONE, IX} \]
- **DONE** \[ \text{HALT} \]

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

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

\[ \begin{array}{c}
\vdots \\
A \\
\vdots \\
\vdots \\
\text{LASTA} \\
\end{array} \]
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 without index regs

<table>
<thead>
<tr>
<th>Operation</th>
<th>With Index Registers</th>
<th>Without Index Registers</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

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

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOADi</td>
<td>N, IX</td>
</tr>
<tr>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>JZi</td>
<td>DONE, IX</td>
</tr>
<tr>
<td>LOAD</td>
<td>LASTA, IX</td>
</tr>
<tr>
<td>ADD</td>
<td>LASTB, IX</td>
</tr>
<tr>
<td>STORE</td>
<td>LASTC, IX</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HALT</td>
</tr>
</tbody>
</table>

- *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></td>
<td>10 (8)</td>
</tr>
<tr>
<td>store</td>
<td></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)** 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></td>
</tr>
</tbody>
</table>

N starts with \(-n\)

A

\[ A_i \]

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

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

<table>
<thead>
<tr>
<th>LOADi</th>
<th>ix</th>
<th>N, IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td></td>
<td>N starts with (-n)</td>
</tr>
<tr>
<td>JZi</td>
<td>done</td>
<td>IX</td>
</tr>
<tr>
<td>LOAD</td>
<td>lasta,</td>
<td>IX</td>
</tr>
<tr>
<td>ADD</td>
<td>lastb,</td>
<td>IX</td>
</tr>
<tr>
<td>STORE</td>
<td>lastc,</td>
<td>IX</td>
</tr>
<tr>
<td>JUMP</td>
<td>loop</td>
<td></td>
</tr>
<tr>
<td>DONE</td>
<td></td>
<td>HALT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>With Index Regs</th>
<th>Without Index Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A</strong></td>
<td></td>
<td>N starts with (-n)</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<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>1</td>
<td>5 (4)</td>
</tr>
</tbody>
</table>
Using Index Registers

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

<table>
<thead>
<tr>
<th>LOADi</th>
<th>N, IX</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOOP</td>
<td></td>
</tr>
<tr>
<td>JZi</td>
<td>DONE, IX</td>
</tr>
<tr>
<td>LOAD</td>
<td>LASTA, IX</td>
</tr>
<tr>
<td>ADD</td>
<td>LASTB, IX</td>
</tr>
<tr>
<td>STORE</td>
<td>LASTC, IX</td>
</tr>
<tr>
<td>JUMP</td>
<td>LOOP</td>
</tr>
<tr>
<td>DONE</td>
<td>HALT</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Instruction Fetch</th>
<th>With Index Regs</th>
<th>Without Index Regs</th>
</tr>
</thead>
<tbody>
<tr>
<td></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>

N starts with \(-n\)

\[A_i \leftarrow A_{i-n} + B_{i-n}, \quad 1 \leq i \leq n\]

\[LASTA\]

- Costs?
Using Index Registers

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

- \textit{Program does not modify itself}
- \textit{Efficiency has improved dramatically (ops / iter)} with index regs vs. without index regs
  - Instruction fetch: 5 (2) vs. 17 (14)
  - Operand fetch: 2 vs. 10 (8)
  - Store: 1 vs. 5 (4)
- \textit{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 $k$

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

new instruction
Operations on Index Registers

To increment index register by $k$

- $AC \leftarrow (IX)$  
- $AC \leftarrow (AC) + k$  
- $IX \leftarrow (AC)$

new instruction

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

To increment index register by k

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

also the AC must be saved and restored

It may be better to increment IX directly

\[ \text{INCI} \quad k, \text{IX} \quad \text{IX} \leftarrow (\text{IX}) + k \]
Operations on Index Registers

To increment index register by k

\[ AC \leftarrow (IX) \]
\[ AC \leftarrow (AC) + k \]
\[ IX \leftarrow (AC) \]

new instruction

also the AC must be saved and restored

It may be better to increment IX directly

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

More instructions to manipulate index register

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

...
Operations on Index Registers

To increment index register by \( k \)

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

also the AC must be saved and restored

It may be better to increment IX directly

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

More instructions to manipulate index register

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

... 

\( IX \) begins to look like an accumulator

\( \Rightarrow \) several index registers

\( \Rightarrow \) several accumulators

\( \Rightarrow \) General Purpose Registers
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
</tr>
<tr>
<td>a2</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

call F

b1

b2

Subroutine F

F:

.     .     .

return
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
<th>a2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>.</td>
<td>.</td>
</tr>
</tbody>
</table>

Subroutine F

return

F:
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a2</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td></td>
<td>.</td>
</tr>
<tr>
<td>call F</td>
<td>b1</td>
</tr>
<tr>
<td></td>
<td>b2</td>
</tr>
</tbody>
</table>

F:

Subroutine F

return
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
<th>a2</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>call F</td>
<td>b1</td>
<td>b2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

F:

Subroutine F

. . .

return
Support for Subroutine Calls

Main Program

<table>
<thead>
<tr>
<th>call F</th>
<th>a1</th>
<th>a2</th>
<th>.</th>
<th>.</th>
</tr>
</thead>
<tbody>
<tr>
<td>call F</td>
<td>b1</td>
<td>b2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Subroutine F

. . .

return
A special *subroutine jump instruction*

A: \texttt{JSR} F

M[F] ← A + 1 and jump to F + 1
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means $AC \leftarrow M[M[x]]$

...
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD  (x) means AC ← M[M[x]]

---

**Events:**

- **S1**  LOAD (F)
- **S2**  STORE (F)
- **S3**  JUMP (F)

---

**Subroutine**

- **F**
- **F+1**

---

**Caller**

- **A**
- **A+3**

---

**JSR** F

- **arg**
- **result**

---

**Graphical Representation:**

- **fetch**
- **arg**
- **store**
- **result**
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)

... 

**Events:**

Execute A

**Subroutine**

F

F+1

S1  LOAD (F)  
    inc F

S2  STORE(F)  
    inc F

S3  JUMP (F)  

**Fetcher**

arg

**Store**

result

**Caller**

A

JSR F

arg

result

A+3
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means $AC \leftarrow M[M[x]]$

... 

**Events:**

Execute A

**Subroutine**

Fetch arg

Store result

**Caller**

A

JSR F

arg

result

A+3

**Subroutine**

F

F+1

S1 LOAD (F)

inc F

S2 STORE(F)

inc F

S3 JUMP (F)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \((x)\) means \(AC \leftarrow M[M[x]]\)

... 

**Events:**

- **A**
  - JSR \(F\)
  - arg
  - result

- **A+3**

**Subroutine**

- **F**
  - \(A+1\)

- **F+1**

- **S1**
  - LOAD \((F)\)
  - inc \(F\)

- **S2**
  - STORE \((F)\)
  - inc \(F\)

- **S3**
  - JUMP \((F)\)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... 

**Events:**

Execute A

Execute S1

**Subroutine**

F

S1 LOAD (F) inc F

S2 STORE(F) inc F

S3 JUMP (F)
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)

...  

**Events:**

- Execute A
- Execute S1

**Subroutine**

- Fetch arg
- Store result

### Caller

- JSR \( F \)
- arg
- result

### Subroutine

- LOAD \( (F) \)
- inc F
- STORE \( (F) \)
- inc F
- JUMP \( (F) \)
Indirect Addressing and Subroutine Calls

Indirect addressing
LOAD (x) means AC ← M[M[x]]
...

Events:
1. JSR F
   - Execute A
2. LOAD (F)
   - Execute S1
   - Inc F
3. STORE (F)
   - Execute S2
   - Inc F
4. JUMP (F)
   - Execute S3
   - Jump to F

Subroutine

Fetch
arg

Store
result

Caller

A

JSR F
arg
result

A + 3
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

Events:

- Execute A
- Execute S1
- Execute S2

---

Figure:

- **Caller**
  - A
  - JSR F
  - arg
  - result
  - A + 3

- **Subroutine**
  - F
  - F + 1
  - S1 LOAD (F)
  - inc F
  - S2 STORE(F)
  - inc F
  - S3 JUMP (F)

- **Fetch**
- **Store**
- **Result**
Indirect Addressing and Subroutine Calls

**Indirect addressing**
LOAD \( (x) \) means \( \text{AC} \leftarrow \text{M}[\text{M}[x]] \)

... [Diagram showing caller and subroutine events]

**Events:**
- Execute A
- Execute S1
- Execute S2

Subroutine fetch
- LOAD \( (F) \)
- inc F

Subroutine store
- STORE(\( F \))
- inc F

Subroutine result
- JUMP(\( F \))
Indirect Addressing and Subroutine Calls

Indirect addressing
LOAD \( (x) \) means \( AC \leftarrow M[M[x]] \)
...

Events:
- Execute A
- Execute S1
- Execute S2
- Execute S3

Subroutine
- fetch arg
- store result

Caller
- JSR F
  - arg
  - result
- A
- A + 3

Subroutine
- F
- F + 1
- S1 LOAD (F)
  - inc F
- S2 STORE(F)
  - inc F
- S3 JUMP (F)
Indirect Addressing and Subroutine Calls

Indirect addressing 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)
Indirect Addressing and Subroutine Calls

*Indirect addressing*

\[
\text{LOAD } (x) \quad \text{means } AC \leftarrow M[M[x]]
\]

\[\ldots\]

**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? ⇒*
Indirect Addressing and Subroutine Calls

**Indirect addressing**

LOAD (x) means AC ← M[M[x]]

... (Diagram showing the process of indirect addressing)

**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**
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 $R_2$. 

- PC
- SP
- ...
- registers
- memory
- Pure Code
- Data
- Stack
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
   
   \[ \text{LOAD} \ x \]

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

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

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

   \[ R \leftarrow M[M[x] + (IX)] \]
   
   or \[ R \leftarrow M[M[x + (IX)]] \]
Evolution of Addressing Modes

1. Single accumulator, absolute address
   
   \texttt{LOAD \ x}

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

3. Indirection
   
   \texttt{LOAD \ (x)}

4. Multiple accumulators, index registers, indirection
   
   \texttt{LOAD \ \text{R, IX, x}}
   
   or\hspace{1cm}
   \texttt{LOAD \ \text{R, IX, (x)}}

   the meaning?

   \begin{align*}
   & \texttt{R} \leftarrow \texttt{M}[\texttt{M}[\texttt{x}] + (\texttt{IX})] \\
   \text{or} & \hspace{1cm} \texttt{R} \leftarrow \texttt{M}[\texttt{x} + (\texttt{IX})]
   \end{align*}

5. Indirect through registers
   
   \texttt{LOAD \ \text{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 ← M[M[x] + (IX)]
   or R ← 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

  \[(\text{Reg op Reg}) \text{ to Reg} \quad R_i \leftarrow (R_j) \text{ op } (R_k)\]
  
  \[(\text{Reg op Mem}) \text{ to Reg} \quad R_i \leftarrow (R_j) \text{ op } M[x]\]

  - 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

  \[(\text{Reg op Reg}) \text{ to Reg} \quad R_i \leftarrow (R_i) \text{ op } (R_j)\]
  
  \[(\text{Reg op Mem}) \text{ to Reg} \quad R_i \leftarrow (R_i) \text{ op } M[x]\]
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

```
load     M[sp] ← 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
  
  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!
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