Unit-2 Computing Platform and Design Analysis

Bus-based Computer Systems

CPU Bus

→ Bus is a set of wires
→ Bus allows CPU, memory, devices to communicate
→ It is a shared communication medium.

Bus Protocols

→ Bus protocol determines how devices communicate.
→ Devices on the bus go through sequences of states.
→ Protocols are specified by state machines, one state machine per actor in the protocol.

Four cycle handshake

1. Device 1 raiseseng
2. Device 2 responds withack
3. Device 2 lowersack once it has finished.
4. Device 1 lowers eng.

Diagram:

```
\[ \text{Device 1} \quad \text{Device 2} \]
\[ \quad \text{ack} \quad \text{eng} \quad \text{ack} \quad \text{eng} \]
\[ \quad 1 \quad 2 \quad 3 \quad 4 \quad \text{time} \]
```
- Clock provided synchronization
- R/W is true when reading
- R/W is false when writing
- Address is a-bit bundle of address lines
- Data is n-bit bundle of data lines
- Data ready signals when n-bit data is ready

State diagrams for bus read:

Get data → Done
→ See ack → Adr's

Wait
CPU

Send data → Release ack

Ack → Adr's

Wait

Start device

Bus Multiplexing

CPU

data enable

device

data

device

Adr's enable

Adr's
Direct Memory Access (DMA) performs data transfer without executing instructions.

- CPU sets up transfer
- DMA engine fetches, writes
- DMA controller is a separate unit.

Bus master
- By default, CPU is bus master & initiates transfers.
- DMA must become bus master to perform its work.
  - CPU can't use bus while DMA operates.
- Bus mastering protocol:
  - Bus request
  - Bus grant.
- DMA operation:
  - Once DMA is bus master, it transfers automatically.
  - May run continuously until complete.
  - May use every n-th bus cycle.
System bus Configuration

- Multiple buses allow parallelism.
- Slow devices on one bus
- Fast devices on separate bus
- Bridge connects buses.

**Memory Components**

- Several different types of memory
  - DRAM
  - SRAM
  - Flash
- Each type of memory comes in varying capacities & widths.

**Address**

- Memory array

- Enable
  - Data
Random Access Memory
- Dynamic RAM is dense, requires refresh
- Synchronous DRAM is dominant type
- It uses clock to improve performance, pipeline memory accesses
- Static RAM is faster, less dense, consumes more power.

Read-only memory
- ROM may be programmed at factory.
- Flash is dominant form of field-programmable ROM
  - Electrically erasable
  - Must be block erased
  - Random access, but write/erase is much slower than read.
  - NOR flash is more flexible
  - NAND Flash is more dense.

Flash Memory
- Non Volatile Memory
  - Flash can be programmed in-circuit
  - Random access for read.
  - To write, Erase a block to 1
  - White bits to 0.

Flash Writing
- Write is much slower than read
  - 1.6 ms write, 70 ns read
- Blocks are large.
Types of flash
1. NOR - Word accessible, read - Erase by blocks
2. NAND - Read by pages, Erase by blocks
3. NAND is cheaper, has faster erase, sequential access times

Timers & Counters
- Timer is incremented by a periodic signal
- Counter is incremented by an asynchronous, occasional signal
- rollers cause interrupt

Watchdog Timer
- It is periodically reset by system
- if watchdog is not reset, it generates an interrupt to reset the host.

- Must use resistor to limit current.
1-segment LCD Display
- May use parallel or multiplexed input
- Types of high resolution display
  - Liquid crystal display (LCD)
  - Plasma, OLED etc.
  - Touchscreen
  - Inclusion of input & output device
  - Input device is a 2D voltmeter
  - DAC
    - Use resistor tree

Flash A/D Conversion
- N bit result requires 2^n comparators

Dual Slope Conversion
- Use counter to time rise to charge/dischage capacitors
- Charging then discharging eliminates non-linearities

\[ V_{in} \rightarrow \text{Encoder} \rightarrow \text{Comparator} \rightarrow \text{DAC} \]
Bus-based Computer Systems

System Architectures

- Components → software & hardware
- Some software is hardware dependent.

Hardware Platform Architecture

- Contains several elements like CPU, bus, memory, I/O devices (networking, sensors, actuators)

Software Architecture

- Functional description must be broken into pieces → 1) division among people
- 2) conceptual organization
- 3) performance 4) testability 5) maintenance

Hardware & Software Architectures

- Hardware & software are intimately related
- Software doesn’t run without hardware
- How much hardware we need is determined by the software requirements → speed, memory

Adding logic to a board:

- Programmable logic devices (PLDs) provide low/medium density logic.
- Field Programmable Gate Array (FPGAs) provide more logic at multi-level logic.
- Application specific integrated circuits (ASICs) are manufactured for a single purpose.
PC as a platform

**Advantages**
- Cheap and easy to get
- Rich and familiar software environment

**Disadvantages**
- Requires a lot of H/W resources
- Not well adapted to real time

**Typical PC H/W platform**

```
CPU

Memory

DMA Controller

Timers

Bus Interface

High speed bus

Device

Bus Interface

Low speed bus

Device
```

**Typical Buses**
- PCI: Standard for high-speed interfacing
- USB, Firewire (IEEE 1394): Relatively low-cost serial interface with high speed

**Software elements**
- IBM PC uses BIOS (Basic I/O System) to implement low-level functions:
  - Boot-up
  - Minimal device drivers
- BIOS has become a generic term for the lowest level system software.
Example: StrongARM

- StrongARM system includes:
  - MPU chip (3.626 MHz clock).
  - System control module (33.768 kHz clock).
  - Real time clock.
  - Operating system time.
  - General purpose I/O.
  - Interrupt controller.
  - Power manager controller.
  - Reset controller.

Debugging embedded systems.

Challenges:

- Target system may be hard to observe.
- Target may be hard to control.
- May be hard to generate realistic inputs.
- Setup sequence may be complex.

Host target design.

Use a host system to prepare s/w for target system.

```
Host System       target system
      |            |
     serial link
```


Host-based tools:

- Cross compile
  - Compiles code on host for target sys.
- Cross debugger
  - Displays target state, allows target system to be controlled.
  - Software debugger
- A monitor program residing on the target provides basic debugger functions.
- Debugger should have a minimal footprint in memory
- User program must be careful not to destroy debugger program, but should be able to recover from some damage caused by user code.

Breakpoints:

- A breakpoint allows the user to stop execution, examine system state, and change state.
- Replace breakpointed instruction with a subroutine call to the monitor program.
In-Circuit Emulators:
- A microprocessor in-circuit emulator is a specially instrumented microprocessor.
- Allows us to stop execution, examine CPU state, modify registers.

Logic Analyzers:
- Logic Analyzer is an array of low-grade oscilloscopes.

Boundary Scan:
- Simplifies testing of multiple chips on a board.
- Registers on pins can be configured as a scan chain.
- Used for debuggers, in-circuit emulators.

Debugging real-time code:
- Bugs in drivers can cause non-deterministic behavior in the foreground problem.
- Bugs may be time dependent.
System level performance analysis.
- Performance depends on all the elements of the system: CPU, Cache, Bus, Main memory, I/O device.

Bandwidth as performance

- Bandwidth applies to several components: memory, bus, CPU fetches.
- Different parts of the system run at different clock rates.
- Different components may have different widths (bus, memory).
- To increase bandwidth:
  - Increase bus width.
  - Increase bus clock rate.

Bus bandwidth:

\[ T = \text{bus cycles} \]
\[ p = \text{time/bus cycle} \]

Total time for transfer:

\[ t = TP \]
\[ w = \text{data payload length} \]
\[ 01 + 0a = \text{overhead} \]

\[ T_{\text{basic}}(N) = (w + 0) N / W. \]
Parallelism
→ speed things up by running several units at once
→ DMA provides parallelism if CPU doesn’t need the bus

```
cpu setup
bus
time
```

```
cpu setup

```
calc 1, 2

```

→ Sequential

```
cpu setup

time
```

```
calc 1
```

```
calc 2
```

→ Parallel

**Program Design & Analysis**

**Software State Machine**

→ state m/c keeps internal state as a variable, changes state based on inputs.

→ Uses - control dominated code - reactive systems

→ Signal processing & circular buffer

→ commonly used in signal processing

→ new data constantly arrives

→ each data has a limited lifetime.
Multiple Module programs.

- Programs may be composed from several files.
- Addresses become more specific during processing.
  - Relative addresses are measured relative to the start of a module.
  - Absolute addresses are measured relative to the start of the CPU address space.

Assemblers

- Major tasks
  - Generate binary for symbolic insts.
  - Translate labels into addresses
  - Handle pseudo-ops.
- Generally one-to-one translation.

Pseudo-operations

- Pseudo-ops do not generate instructions:
  - ORG sets program location.
  - EQU generates symbol table entry without advancing PEA.
  - DATA statements define data blocks.

Linking

- Combines several object modules into a single executable module
  - Put modules in order
  - Resolve labels across modules.
Module ordering

→ Code modules must be placed in absolute positions in the memory space.
→ Loader map or linker flags control the order of modules.

Dynamic linking.

→ Some operating systems link modules dynamically at run time.
→ Shares one copy of library among all executing programs.
→ Allows programs to be updated with new versions of libraries.

Program Design & Analysis.

→ Compilation = translation + optimization.
→ Compiler determines quality of code:
  → Use of CPU resources
  → Memory access scheduling
  → Code size

Statement translation & Optimization.

→ Source code is translated into intermediate form such as CDFG
→ CDFG is transformed/optimized
→ CDFG is translated into instructions with optimization decisions.
→ Instructions are further optimized.
Arithmetic expressions:

\[ a \times b + \frac{s \times (c - d)}{a} \]

Control Code Generation:

\[ \text{if } (a + b > 0) \]
\[ x = 5; \]
\[ \text{else} \]
\[ x = 7; \]

Procedure Linkage:

- Need code to call & return
- Pass parameters & results
- Parameters & returns are passed on stack
- Procedures with few parameters may use registers.
Data structures

Different types of data structures use different data layouts.

Some offsets into data structure can be computed at compile time, others must be computed at run time.

One dimensional arrays.

- c array name points to 0th element

```
  a →
  a[0]
  a[1]
  a[2]
```

Two dimensional arrays

```
  a[0,0]
  a[0,1]
  a[0,2]
  a[1,0]
  a[1,1]
```

Loop transformations

Goals
- reduce loop overhead
- increase opportunities for pipelining
- improve memory sys performance

Loop Tiling
- Breaks one loop into a nest of loops
- Changes order of accesses within loop
- Changes cache behaviour
```java
for (i = 0; i < N; i++)
    for (j = 0; j < N; j++)
        c[i][j] = a[i][j] * b[i][j];

for (ii = 0; i < N; i++)
    for (jj = j; j < min(i + x, N); jj++)
        e[i][j] = a[round(i)] * b[i][j];
```

Array Padding

-> Add array elements to change mapping into cache.

Before

\[
\begin{array}{ccc}
    a[0,0] & a[0,1] & a[0,2] \\
    a[1,0] & a[1,1] & a[1,2] \\
\end{array}
\]

After

\[
\begin{array}{ccc}
    a[0,0] & a[0,1] & a[0,2] \\
    a[1,0] & a[1,1] & a[1,2] \\
\end{array}
\]

Registered Allocation

1) Choose register to hold each variable
2) Determine lifespan of variable in the register.

Instruction Scheduling

- Non pipelined m/c do not need
  inits. scheduling.
- In pipelined m/c, execution time of
  one instr. depends on the nearby
  inits: opcodes, operands.
Software pipelining
- schedules instructions across loop iterations.
- Reduces instruction latency in iteration i by inserting instructions from iteration i+1.

Interpreters vs. JIT compilers.
Interpreters -> translates & executes program statements.
JIT compilers -> compile small sections of code into instructions during program execution.

Program design & analysis.
- Program level performance analysis:
  1. Optimizing for execution time
  2. Energy / Power
  3. Program size
  4. Program validation & testing.

Complexities of program performance:
- Varies with input data
- Cache effects
- Instruction level performance variations.
How to measure program performance

- Simulate execution of the CPU.
- Measure on real CPU using time.
  - Requires modifying the program to control the time.
  - Measure on real CPU using logic analyzer.
  - Requires events visible on the pins.

Program performance metrics

- Average-case execution time
  - Used in application programming
- Worst-case execution time
  - Component in deadline satisfaction
- Best-case execution time
  - Task level interactions can cause best-case program behavior to result in worst case system behavior.

Elements of program performance

- Program execution time = program path + instruction timing.
- Solving these problems independently helps simplify analysis.
- Accurate performance analysis requires 1. Assembly/binary code
  2. Execution platform.
Instruction Timing

→ Not all instructions take the same amount of time.
→ Execution times of instructions are not independent.
→ Execution times may vary with operand value.
   1) Floating point operations
   2) Multi-cycle integer operations

Measurement driven performance analysis

→ Must actually have access to the CPU.
→ Must know data inputs that give worst/best case performance.
→ Must make state visible.

Trace-driven measurement
→ Trace files are large.
→ Requires modifying the program.
→ Widely used for cache analysis.

Physical Measurement
→ In-circuit emulator allows tracing.
→ Affects execution timing.
→ Logic Analyzer can measure behavior at pins.
→ Address bus can be analyzed to look for events.
CPU simulation

→ Some simulators are less accurate.
→ Cycle-accurate simulator provides accurate cycle-clocked timing.

Performance optimization Motivation.

→ Embedded systems often meet deadlines.
→ Need to be able to analyze execution time.
→ Need techniques for reliably improving execution time.

Loop Optimizations

→ Loops are good targets for optimization.
→ Basic loop optimizations:
  1. Code motion
  2. Induction variable elimination
  3. Strength reduction (x^2 \rightarrow x < 1)

Cache Analysis

→ Loop nest - set of loops, one inside the other.
→ Perfect loop nest - no conditionals in nest.
→ Because loops use large quantities of data, cache conflicts are common.
Performance optimization

→ Use registers efficiently
→ Use page mode memory access
→ Analyze cache behavior.

Energy/Power Optimization

→ Energy: ability to do work
  Most important in battery powered systems.
→ Power: energy per unit time
  Important even in wall-plug systems - power becomes heat.

Efficient loops

→ Don't use function calls
→ Keep loop body small to enable local repeat (only forward branches)
→ Use unsigned integer for loop counter
→ Use <= to test loop counter
→ Make use of compiler - global optimization, software pipelining.

Optimizing for program size:

→ reduce flow cost of memory
→ reduce power consumption of memory units.
→ 2 opportunities: (data, instructions).
Data size minimization:

- Reuse constants, variables, data buffers in different parts of code.
- Requires careful verification of correctness.
- Generate data using instructions.

Reducing code size:

- Avoid function inlining.
- Choose CPU with compact instructions.
- Use specialized instructions where possible.

Program validation and testing:

Testing strategies:

1. Black box: doesn’t look at the source code.
2. Clear box (white box) does look at the source code.

Clear-box testing:

Testing procedure:

- Controllability: provide program with inputs.
- Execute.
- Observability: examine outputs.

Execution paths & testing:

- Paths are important in functional testing as well as performance analysis.
In general, an exponential no. of paths through the program.

Choosing the paths to test

- Execute every statement at least once.
- Execute every branch direction at least once.

Branch testing

- Exercise true and false branches of conditional.
- Exercise every simple condition at least once.

Loop testing

- Loops need specialized tests to be tested efficiently.

Black-box testing

- Complements clear box testing
- Tests software in different ways.

Black-box test vectors

- Random tests
- Regression tests.
Designing with computing platforms

1. Example platforms
   - Evaluation boards
     - Designed by CPU manufacturers or others
       → Include CPU, memory, some I/O devices
       → May include prototyping section
       → CPU manufacturers often give out reference design - can be used as starting point for our custom board design
     - ARM evaluation module includes ARM processor, display, serial port & prototyping area
   - Beagle Board - OMAP processor, Audio input, output, video output & SD card

2. Choosing a platform
   - Hardware
     - CPU: choice of instruction sets, features etc.
     - Bus: determines available I/O devices
     - System performance
     - Memory: size & speed
     - I/O devices: vary in performance, cost
   - Software
     - Run Time Components & Support Components
   - Intellectual Property
     - Hardware designs, source or object code
- Used at all levels of design:
  - Schematics for hardware reference design
  - Drivers & run-time libraries
  - Software development environments

- Beagle Board IP
- PCB schematics & artwork files
- Bill of materials for components
- Compiler
- Simulator

**Debugging embedded systems**

**Challenges**
- Target system may be hard to observe.
- Target may be hard to control.
- May be hard to generate realistic inputs.
- Setup sequence may be complex.

**Host/target design**
- Use a host system to prepare software for target system.

![Diagram of Host System and Target System](attachment:diagram.png)

- Use serial line to communicate between host and target systems.
- Host uses tools:
  - Cross compiles code on host for target system.
  - Cross debugger displays target state, allows target system to be controlled.
Debugging Techniques

Software Debuggers

- A monitor program residing on the target provides basic debugger function.
- Debugger should have a minimal footprint in memory.
- User program must be careful not to destroy debugger program, but should be able to recover from some damage caused by user code.

Breakpoints

- A breakpoint allows the user to stop execution, examine system state, and change state.
- Replace the breakpointed instruction in the program with a subroutine call to the monitor program.

```
0x400: MUL x4, x4, x6
0x404: ADD x2, x2, x4
0x408: ADD y0, y0, #1
0x40c: BL breakpoint
```
Breakpoint handler actions:
- Save registers
- Allow user to examine machine
- Before returning, restore system state. Safest way to execute the instruction is to replace it and execute in place.
- Put another breakpoint after the replaced breakpoint to allow restoring the original breakpoint.

In-Circuit Emulators:
- Microprocessor in-circuit emulators are specially instrumented.
- Allows us to stop execution, examine CPU state, modify registers.

Logic Analyzers - is an array of low-grade oscilloscopes.

Logic Analyzer Architecture:

```
<table>
<thead>
<tr>
<th>System Data</th>
<th>Sampled Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Clock</td>
<td>Controller</td>
</tr>
<tr>
<td></td>
<td>Display</td>
</tr>
<tr>
<td></td>
<td>Keypad</td>
</tr>
</tbody>
</table>
```

Microprocessor

Vector Address
Boundary scan
- Simplifies testing of multiple chips on a board.
- Registers on pins can be configured as a scan chain.
- Used for debuggers, in-circuit emulators.

How to exercise code
- Run on host system
- Run on target system
- Run in instruction-level simulator.
- Run on cycle-accurate simulator.
- Run in hardware/software co-simulation environment.

Debugging real-time code
- Bugs in drivers can cause non-deterministic behavior in the foreground problem.
- Bugs may be timing-dependent.

Consumer Electronics Architecture

Use Functional Requirements
- Multimedia: stored in compressed form, uncompressed when viewing.
- Data storage & management: keep track of multimedia.
- Communication: download, upload, chat

Non-functional Requirements
- Battery operated
- Strict power budget
- Inexpensive
- User interface must be capable
  Use case for playing Multimedia

Use case for synchronizing with a host system

Hardware Architecture
- I/O devices
  - CPU
  - DSP
  - Network
  - Storage
File Systems
- Flash is used for mass storage.
- Flash wears out on writing (upto 1 million cycles)
  -> Directory most often written, wears out first.
- Flash File system has layer that moves contents to levelize wear.
  -> Hides wear leveling from API
- Platform level performance analysis.
  -> Performance depends on all the elements of the system: CPU, Cache, Bus, Main Memory, I/O device.

Bandwidth or performance.
- Bandwidth applies to several components: Memory, Bus, CPU fetches.
- Different parts of the system run at different clock rates.
- Different components may have different widths (bus, memory)
- Bandwidth & data transfer
  - Video frame: $320 \times 240 \times 3 = 230400$ bytes
Transfer in 1/30 sec.
- Transfer 1 byte/μsec, 0.33 sec per frame
  - Too slow.
- Increase bandwidth:
  - Increase bus width, bus clock rate.

**Bus Bandwidth**

\[ T = \text{Bus cycles} \]
\[ P = \text{Time/bus cycle} \]

Total time for transfer: \( t = TP \)

\( D \): data payload length.

Overhead \( O = 01 + 02 \)

Bus burst transfer bandwidth:

\[ T_{\text{basic}}(N) = \frac{(D+O)N}{W} \]

\[ T_{\text{burst}}(N) = \frac{(BD+O)N}{BW} \]

**Memory Aspect Ratios**

- Memory Access Times:
  - Memory component access times come from chip data sheet.
  - Page mode allows faster access for successive transfers on same page.
  - If data doesn’t fit naturally into physical words:
    \[ A = \left\lfloor \frac{E}{W} \right\rfloor \mod W + 1 \]
Program level performance analysis.

- Need to understand performance in detail:
  - Real time behavior, not just typical.
  - On complex platform.
- Program performance ≠ CPU performance.
  - We must analyze the entire program.

Complexities of program performance:
- Varies with input data
  - Different length paths.
- Cache effects.
- Instruction level performance variations:
  - Pipeline interlocks.
  - Fetch times.

How to measure program performance:
- Simulate execution of the CPU:
  - Make CPU state visible.
- Measure on real CPU using timers:
  - Requires modifying the program to control the time.
- Measure on real CPU using logic analyser.
Program performance metrics:
- Average case execution time:
  typically used in application programming.
- Worst case execution time:
  A component in deadline satisfaction.
- Best case execution time:
  Task-level interactions can cause best-case program behaviors to result in worst-case system behaviors.

Elements of program performance:
Execution Time = Program path + Instruction Timing.
Accurate performance analysis requires:
- Assembly/Binary Code
- Execution Platform.

Data dependent paths in if statements:
```c
if (a < 10) { b = 1; }
if (c) { y = 1; }
else if (d) { x = y; }
else { z = x + y; }
```

### Table

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>path</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>T1=T3=F; No assignments</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>T1=T; T3=T; A4</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>T1=T; T2=F; A2, A3</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>T1=T2=T; A1, A3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>T1=T; T2=F; A2, A3</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>T1=T2=T; A1, A3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>T1=T; T2=F; A2, A3</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T1=T2=T; A1, A3</td>
</tr>
</tbody>
</table>

**Paths in a loop**

for (i=0, j=0; i<N; i++)

\[ f = f + x[i] * e[i]; \]

- Instruction Timing
  - Not all instructions take the same amount of time.
  - Execution times of instructions are not independent.
  - Execution times may vary with operand value.
Measurement driven performance analysis
- Not so easy.
- Must actually have access to CPU.
- Must know data inputs that give worst/best case performance.
- Must make state visible.
- Still an important method for performance analysis.

Feeding the program.
- Need to know the desired input values.
- May need to write software scaffolding to generate the input values.
- Software scaffolding may also need to examine outputs to generate feedback-driven inputs.

Trace-driven Measurement.
- Trace-driven
  - Instrument the program.
  - Save information about the path.
- Requires modifying the program.
- Trace files are large.
- Widely used for cache analysis.
Physical Measurement
- In-circuit emulator allows tracing.
  - Affects execution timing
- Logic analyzer can measure behavior at points.
  - Address bus can be analyzed to look for events.
  - Code can be modified to make events visible.
- Particularly important for real-world input streams.

CPU Simulation
- Some simulators are less accurate.
- Cycle-accurate simulator provides accurate clock cycle timing.
  - Simulator models CPU internals
  - Simulator writer must know how CPU works.

Performance Optimization Strategies
- Embedded systems must often meet deadlines.
  - Faster may not be fast enough.
- Need to be able to analyze execution time.
  - Worst case, not typical.
- Need techniques for reliably improving
Programs & performance analysis
- Best results come from analyzing optimized instructions, not high level language code.
  - Cache effects are hard to predict.
  - Code motion.

Loop Optimizations
- Loops are good targets for optimization.
- Basic loop optimizations:
  - Code motion.
  - Induction-variable elimination.
  - Strength reduction.

Code Motion
for (i=0; i<=N*M; i++)
  z[i] = a[i] + b[i];

\[
\begin{align*}
v &= 0; \\
x &= N \times M \\
\end{align*}
\]

\[
\begin{align*}
\text{if } i < x & \\
\text{then } z[i + 1] &= a[i] + b[i] \\
\text{else } z[i + 1] &= a[i] + b[i] \\
\text{end if} \\
\text{if } i < x & \\
\text{then } i &= i + 1 \\
\text{else } i &= i + 1 \\
\text{end if}
\end{align*}
\]
Cache Analysis.

Loop nest: set of loops, one inside other.

Perfect loop nest: no conditionals in nest.
- Because loops use large quantities of data, cache conflicts are common.

Performance Optimization:
- Use registers efficiently.
- Use page mode memory accesses.
- Analyze cache behavior:
  - Instruction conflicts can be handled by rewriting code, rescheduling;
  - Conflicting scalar data can easily be moved.
  - Conflicting array data can be moved, padded.