Final Exam Review Slides

Fall 2017
Review Topics

- Number Representation
- C Programming
- LC-3 ISA, Programming, Pointers/Stack/Heap

Logic:
- Transistors/Gates,
- Boolean algebra/Combinational Logic
- Sequential Logic

LC-3 dataflow and control

Architecture:
- Parallelism and performance
- Memory hierarchy
Number Representation: Conversions

• Decimal ↔ Binary ↔ Hexadecimal
• Signed binary numbers: conversion/add/subtract/sign extend
Number Representation

Binary to Floating Point Conversion

- Single-precision IEEE floating point number:
  \[ 1 \ 01111110 \ 100000000000000000000000 \]
  - Or 0xBF400000
  - Sign is 1 – number is negative.
  - Exponent field is 01111110 = 126 – 127 = -1 (decimal).
    - Fraction is 1.10000000000… = 1.5 (decimal).

- Value = \(-1.5 \times 2^{(126-127)} = -1.5 \times 2^{-1} = -0.75\)
Value = 4.25

- Number is positive – sign is 0
- Fraction is 100.01 (binary), normalize to 1.0001 * 2^2
- Exponent is 2 + 127 = 129 (decimal) = 10000001

Single-precision IEEE floating point number:

0 10000001 0001000000000000000000000

or 0x40880000

sign  exponent  fraction
C Programming
C Programming: **Operators**

```c
int i = 0x11223344;   int j = 0xFFFF0000;

C code with bitwise operators:
``` printf("0x%x\n", i & j); 0x11220000
``` printf("0x%x\n", i && j); 0x1

C code with arithmetic operators:
``` int i = 10;   int j = 2;
``` printf("d\n", i / j); 5
```
C Programming

Control Structures

C conditional and iterative statements

- **if statement**
  ```c
  if (value == 0x12345678)
      printf("value matches 0x12345678\n");
  ```

- **for loop**
  ```c
  for (int i = 0; i < 8; ++i)
      printf("i = %d\n", i);
  ```

- **while loop**
  ```c
  int j = 6;
  while (j--)
      printf("j = %d\n", j);
  ```
C Programming
Pointers and Arrays

C pointers and arrays

```c
void foo(int *pointer)
{
    *(pointer+0) = pointer[2] = 0x1234;
    *(pointer+1) = pointer[3] = 0x5678;
}

int main(int argc, char *argv[])
{
    int array[] = {0, 1, 2, 3};
    foo(array);
    for (int i = 0; i <= 3; ++i)
    {
        printf("array[%d] = %x\n", i, array[i]);
    }
}
```
C data structures

// Structure definition
struct sCoordinate
{
    float X;
    float y;
    float z;
};

typedef struct {
    ...
} Coordinate;
C data structures

// Structure allocation
struct sCoordinate coordinates[10]; // no typedef
typedef Coordinate coordinates[10]; // typedef
Coordinate *coordinates = malloc(sizeof(Coordinate)*10);

// Structure access
coordinates[5].X = 1.0f;
pCoordinate->X = 1.0f;
C Programming

Strings

C strings

```c
char *string = "hello";
char *carray = { 'h', 'e', 'l', 'l', 'o' };
char label[20];
strcpy(label, "hello");
strcat(label, " world");
printf("%s\n", label);  // hello world
printf("%d\n", strlen(label));  // 11
printf("%d\n", sizeof(label));  // 20
```
C include files

#include <stdio.h> - FILE, stdout, stdin, stderr, putchar, getchar, printf, scanf, fprintf, fscanf, fopen, fclose, ...
#include <stdlib.h> - atof, atoi, malloc, free, rand, exit, getenv, system, ...
#include <stdbool.h> - bool, true, false
#include <string.h> - memcpy, memset, strcpy, strcat, strlen, strtok, ...
#include <math.h> - sin, cos, tan, exp, log, fmod, fabs, floor, ceil, ...
Functions

C function prototypes must precede implementation of function

```c
int addInt(int i, int j);
float addFlt(float u, float v);
void addInt(int param0, int param1, int *result);
void addFlt(float f0, float f1, float *result);
bool writeFile(char *filename, Instructions[]);
void input(Instruction *pInstruction);
char *printInt(int number);
```
C Programming

Main Program

- C include files
- command line arguments are passed to main
- arguments are strings, may need to convert
- getenv function queries environment variables

```c
#include <stdio.h>
#include <string.h>

int main(int argc, char *argv[])
{
    printf("%d\n", argc); // # of arguments
    printf("%s\n", argv[0]); // program name
    printf("%s\n", argv[1]); // first argument
    printf("%s\n", argv[2]); // second argument
}
```
LC-3 ISA
Addressing Modes

Load -- read data from memory to register
- **LD**: PC-relative mode
- **LDR**: base+offset mode
- **LDI**: indirect mode

Store -- write data from register to memory
- **ST**: PC-relative mode
- **STR**: base+offset mode
- **STI**: indirect mode

Load pointer: compute address, save in register
- **LEA**: immediate mode
  - *does not access memory*
What is the machine code for assembly instruction \texttt{NOT R7,R6}?

Step 1) Identify opcode: \texttt{NOT} = \texttt{1001}

Step 2) Put values into each field:

<table>
<thead>
<tr>
<th>Field</th>
<th>OPCODE</th>
<th>DR</th>
<th>SR</th>
<th>5:0</th>
</tr>
</thead>
<tbody>
<tr>
<td>15:12</td>
<td>111111</td>
<td>11</td>
<td>110</td>
<td>111111</td>
</tr>
<tr>
<td>11:9</td>
<td>1001</td>
<td>111</td>
<td>110</td>
<td>111111</td>
</tr>
<tr>
<td>8:6</td>
<td></td>
<td>111</td>
<td>110</td>
<td>111111</td>
</tr>
</tbody>
</table>

Step 3) Build machine instruction: \texttt{1001111110111111}
LC-3 Architecture
Assembly Code Syntax

.ORIG x3000

MAIN    AND    R0,R0,#0     ; Initialize Sum
JSR COMPUTE       ; Call function
ST     R0, SUM      ; Store Sum
HALT                ; Program complete

COMPUTE  LD     R1,OPERAND1  ; Load Operand1
LD     R2,OPERAND2  ; Load Operand2
ADD    R0,R1,R2     ; Compute Sum
RET                 ; Function return

;; Input data set
OPERAND1 .FILL     x1234     ; Operand1
OPERAND2 .FILL     x4321     ; Operand2
SUM      .BLKW     1         ; Sum
.END
Memory Model

Push and Pop Stack

Assume POP and PUSH code as follows:

MACRO PUSH(reg)
    ADD R6, R6, #1 ; Decrement SP
    STR reg, R6, #0 ; Store value
END

MACRO POP(reg)
    LDR reg, R6, #0 ; Load value
    ADD R6, R6, #1 ; Increment SP
END
Memory Model: Detailed Example

Main program to illustrate stack convention:

```assembly
.ORIG x3000
MAIN   LD R6,STACK    ; init stack pointer
LD R1,OPERAND1 ; load second operand
PUSH R1       ; PUSH second operand
LD R0,OPERAND0 ; load first operand
PUSH R0       ; PUSH first operand
JSR FUNCTION ; call function
LDR R0,R6,#0  ; POP return value
ADD R6,R6,#3  ; unwind stack
ST  R0,RESULT ; store result
HALT
```

<table>
<thead>
<tr>
<th>Local Variable</th>
<th>Frame Pointer</th>
<th>Return Address</th>
<th>Return Value</th>
<th>First Operand</th>
<th>Second Operand</th>
</tr>
</thead>
</table>
Memory Model

Function code to illustrate stack convention:

```
FUNCTION
    ADD R6, R6, #1 ; alloc return value
    PUSH R7        ; PUSH return address
    PUSH R5        ; PUSH frame pointer
    ADD R5, R6, #1 ; FP = SP-1

    ADD R6, R6, #1 ; alloc local variable
    LDR R2, R5, #4 ; load first operand
    LDR R3, R5, #5 ; load second operand
    ADD R4, R3, R2 ; add operands
    STR R4, R5, #0 ; store local variable

    stack exit code
    STR R4, R5, #3 ; store return value
    ADD R6, R5, #1 ; SP = FP+1
    POP R5         ; POP frame pointer
    POP R7         ; POP return address
    RET            ; return
```

Stack before STR instruction

<table>
<thead>
<tr>
<th>FP[0]</th>
<th>Local Variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>FP[1]</td>
<td>Frame Pointer</td>
</tr>
<tr>
<td>FP[2]</td>
<td>Return Address</td>
</tr>
<tr>
<td>FP[3]</td>
<td>Return Value</td>
</tr>
<tr>
<td>FP[4]</td>
<td>First Operand</td>
</tr>
<tr>
<td>FP[5]</td>
<td>Second Operand</td>
</tr>
</tbody>
</table>
Hardware
Transistors and Gates

NOR Gate

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
<td>Open</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>Closed</td>
<td>Open</td>
<td>Open</td>
<td>Closed</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>Open</td>
<td>Closed</td>
<td>Closed</td>
<td>Open</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>Open</td>
<td>Open</td>
<td>Closed</td>
<td>Closed</td>
<td>0</td>
</tr>
</tbody>
</table>
Combinational Logic

Combinational Circuit to Truth Table

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>V</th>
<th>W</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>6</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean algebra:
Some Useful Identities for simplification

\[ AB + AB = A \]

Proof: \[ AB + AB = A(B + B) \]
\[ = A \]

\[ A + AB = A \]

Proof: \[ A + AB = A(1 + B) \]
\[ = A \]
Functional Blocks

2-bit decoder

4-to-1 MUX
Functional Blocks: Full Adder

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C_{in}</th>
<th>S</th>
<th>C_{out}</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Boolean logic lets us reduce the circuit

- $X = A'B'C' + A'BC' + ABC' + ABC = A'C' + AB$
- $Y = A'B'C + A'BC + AB'C + ABC = A'C + AC = C$

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Combinational vs. Sequential

Combinational Circuit

- always gives the same output for a given set of inputs
  - ex: adder always generates sum and carry, regardless of previous inputs

Sequential Circuit

- stores information
- output depends on stored information (state) plus input
  - so a given input might produce different outputs, depending on the stored information
- example: ticket counter
  - advances when you push the button
  - output depends on previous state
- useful for building “memory” elements and “state machines”
Storage Elements

• Static (SRAM): use a circuit with feedback to save a bit of information
  • flipflops
  • Static memories

• Dynamic (DRAM): Use charge at a node to represent a 1 or 0
  • A cell in a dynamic memory
  • Fewer transistors hence cheaper
  • Need periodic refreshing, every few millisecs.

• Both are volatile.

• Not considered here:
  • ROM (read only memory): combinational
  • Flash memory: semiconductor, but work like disks
R-S Latch Summary

R = S = 1
  • hold current value in latch

S = 0, R=1
  • set value to 1

R = 0, S = 1
  • set value to 0

R = S = 0
  • both outputs equal one
  • final state determined by electrical properties of gates
  • *Don’t do it!*
Memory

Now that we know how to store bits, we can build a memory – a logical $k \times m$ array of stored bits.

Address Space:
number of locations (usually a power of 2)

Addressability:
number of bits per location (e.g., byte-addressable)
Flip-flops

D Flip-flop: a storage element, can be edge-triggered (available in logisim)

Rising edge: input sampled

<table>
<thead>
<tr>
<th>D</th>
<th>Next Q</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

State Q is always available
Registers

A register is a row of storage element.

Register with parallel load with a Load control line

Clock is often implied
State Machine

A general sequential circuit

• Combines combinational logic with storage
• “Remembers” state, and changes output (and state) based on inputs and current state
Example 1: Analyze this FSM

Combinational block
In: $x, A, B$ Out: $DA, DB$

$DA = \overline{x}A + AB + x\overline{AB}$
$DB = \overline{x}B + x\overline{B}$

Input: $x$
State: $A, B$
Output: $A, B$
Example 1: Analyze this FSM (last)

State Table

<table>
<thead>
<tr>
<th>Input</th>
<th>Present State</th>
<th>Next State</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>X</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

It is an up counter
Example 2: Danger Sign design from P&P

A blinking traffic sign

• No lights on
• 1 & 2 on
• 1, 2, 3, & 4 on
• 1, 2, 3, 4, & 5 on
• (repeat as long as switch is turned on)

• Input: Switch
• Outputs: Z, Y, X
• States: 4 (bits S1, S0)
• Choose: output depends on the state
Traffic Sign: State Diagram/Table/Design

Minimized implementation
\[ DS1 = \text{In} \sim S1 \ S0 + \text{In} \ S1 \sim S0 \]
\[ DS0 = \text{In} \sim S0 \]
\[ Z = S0 + S1 \]
\[ Y = S1 \]
\[ X = S1 \ S0 \]

<table>
<thead>
<tr>
<th>Input</th>
<th>Present State</th>
<th>Next State</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>In</td>
<td>S1 S0</td>
<td>S1 S0</td>
<td>ZYX</td>
</tr>
<tr>
<td>0 0</td>
<td>0 0</td>
<td>0 0</td>
<td>000</td>
</tr>
<tr>
<td>0 1</td>
<td>0 1</td>
<td>0 0</td>
<td>100</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>111</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>0 1</td>
<td>000</td>
</tr>
<tr>
<td>1 0</td>
<td>0 0</td>
<td>1 0</td>
<td>100</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>1 1</td>
<td>110</td>
</tr>
<tr>
<td>1 1</td>
<td>1 1</td>
<td>0 0</td>
<td>111</td>
</tr>
</tbody>
</table>

State bit S₁
State bit S₀
Outputs
LC-3 Control Architecture

Control FSM: Design a FSM with given outputs.

<table>
<thead>
<tr>
<th>Signal Name</th>
<th>Signal Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD.MAR/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.MDR/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.IR/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.BEN/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.REG/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.CC/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.PC/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.Priv/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.SavedSSP/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.SavedUSP/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>LD.Vector/I:</td>
<td>N0, LOAD</td>
</tr>
<tr>
<td>GatePC/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GateMDR/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GateALU/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GateMARMUX/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GateVector/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GatePC-I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GatePSR/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>GateSP/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>PCMUX2/I:</td>
<td>PC+1, BUS</td>
</tr>
<tr>
<td></td>
<td>select value from bus</td>
</tr>
<tr>
<td></td>
<td>ADDER, select output of address adder</td>
</tr>
<tr>
<td>DRMUX2/I:</td>
<td>11.9, DESTINATION IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>R7, DESTINATION R7</td>
</tr>
<tr>
<td></td>
<td>SP, DESTINATION R6</td>
</tr>
<tr>
<td>SIOMUX2/I:</td>
<td>11.9, SOURCE IR[11:9]</td>
</tr>
<tr>
<td></td>
<td>A.6, SOURCE IR[8:6]</td>
</tr>
<tr>
<td></td>
<td>SP, SOURCE R6</td>
</tr>
<tr>
<td>ADDR1MUX/I:</td>
<td>PC, BaseR</td>
</tr>
<tr>
<td>ADDR2MUX/I:</td>
<td>ZERO, select the value zero</td>
</tr>
<tr>
<td></td>
<td>offset9, select SEXTI(R5:0)</td>
</tr>
<tr>
<td></td>
<td>PCOffset9, select SEXTI(R8:0)</td>
</tr>
<tr>
<td></td>
<td>PCOffset11, select SEXTI(18:0)</td>
</tr>
<tr>
<td>SPMUX2/I:</td>
<td>SP+1, select stack pointer+1</td>
</tr>
<tr>
<td></td>
<td>SP-1, select stack pointer-1</td>
</tr>
<tr>
<td></td>
<td>Saved SSP, select saved Supervisor Stack Pointer</td>
</tr>
<tr>
<td></td>
<td>Saved USP, select saved User Stack Pointer</td>
</tr>
<tr>
<td>MARMUX/I:</td>
<td>7.0, ADDER</td>
</tr>
<tr>
<td></td>
<td>select ZEXTI(R7:0)</td>
</tr>
<tr>
<td>VectorMUX/I:</td>
<td>INTV, Priv. exception</td>
</tr>
<tr>
<td></td>
<td>0pc, exception</td>
</tr>
<tr>
<td>PSRMUX/I:</td>
<td>Individual settings, BUS</td>
</tr>
<tr>
<td>ALUK2/I:</td>
<td>ADD, AND, NOT, PASSA</td>
</tr>
<tr>
<td>MI0.ENC/I:</td>
<td>N0, YES</td>
</tr>
<tr>
<td>R.W/I:</td>
<td>RD, WR</td>
</tr>
<tr>
<td>Set.Priv/I:</td>
<td>0, Supervisor mode</td>
</tr>
<tr>
<td></td>
<td>1, User mode</td>
</tr>
</tbody>
</table>
FSM Description

Instruction Fetch:
S18: MAR<-PC, PC<-PC+1,
   If no INT, go to S33

To implement
MAR<-PC needs: GatePC =1, LD.MAR = 1,
PC = PC+1 needs: PCMUX select PC+1, LD.PC =1

For each state
• activate or select appropriate Control signals
• Go to appropriate next state
Even Littler Computer 3
Computer Architecture
**Processor Execution time**

The time taken by a program to execute is the product of:

- Number of machine instructions executed
- Number of clock cycles per instruction (CPI)
- Single clock period duration

**Clock Cycles = Instruction Count \( \times \) Cycles per Instruction**

**CPU Time = Instruction Count \( \times \) CPI \( \times \) Clock period**

**Example:** 10,000 instructions, CPI=2, clock period = 250 ps

\[
\text{CPU Time} = 10,000 \text{ instructions} \times 2 \times 250 \text{ ps} \\
= 10^4 \times 2 \times 250 \times 10^{-12} = 5 \times 10^{-6} \text{ sec.}
\]
Pipelining Analogy

Pipelined laundry: overlapping execution

- Parallelism improves performance

- Four loads:
  - time
  \[= 4 \times 2 = 8 \text{ hours}\]

- Pipelined:
  - Time in example
  \[= 7 \times 0.5 = 3.5 \text{ hours}\]
  - Non-stop
  \[= 4 \times 0.5 = 2 \text{ hours}.\]
Instruction level parallelism (ILP):

Pipelining is one example.

Multiple issue: have multiple copies of resources

- Multiple instructions start at the same time
- Need careful scheduling
  - Compiler assisted scheduling
  - Hardware assisted ("superscaler"): "dynamic scheduling"
    - Ex: AMD Opteron x4
    - CPI can be less than 1!
Task Parallelism

Program is divided into tasks that can be run in parallel

Example: A program needs subtasks A, B, C, D. B and C can be run in parallel. They each take 200, 500, 500 and 300 nanoseconds.

Without parallelism: total time needed = 200 + 500 + 500 + 300 = 1500 ns.

With Task level parallelism: 200 + 500 (B and C in parallel) + 300 = 1000 ns.
Review: Major Components of a Computer

<table>
<thead>
<tr>
<th>Level</th>
<th>Access time</th>
<th>Size</th>
<th>Cost/GB</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>0.25 ns</td>
<td>48+ 64,128, 32b</td>
<td>-</td>
</tr>
<tr>
<td>Cache L1, L2, L3</td>
<td>0.5 ns</td>
<td>5MB</td>
<td>125</td>
</tr>
<tr>
<td>Memory</td>
<td>1000 ns</td>
<td>8GB</td>
<td>4.0</td>
</tr>
<tr>
<td>SSD</td>
<td>100K ns</td>
<td>0.25</td>
<td></td>
</tr>
<tr>
<td>Disk</td>
<td>1000K ns</td>
<td>1 TB</td>
<td>0.02</td>
</tr>
</tbody>
</table>
Average Access Time

- Hit time is also important for performance

**Average memory access time (AMAT)**

- AMAT = Hit time + Miss rate \( \times \) Miss penalty

**Example:** CPU with cache and main memory

- CPU with 1ns clock, hit time = 1 cycle, miss penalty = 20 cycles, L-cache miss rate = 5%
- AMAT = 1 + 0.05 \( \times \) 20 = 2ns
  - 2 cycles per instruction
Virtual vs. Physical Address

Processor assumes a certain memory addressing scheme:
- A block of data is called a virtual page
- An address is called virtual (or logical) address

Main memory has a different addressing scheme:
- Real memory address is called a physical address, MMU translates virtual address to physical address
- Complete address translation table is large and must therefore reside in main memory
- MMU contains TLB (translation lookaside buffer), which is a small cache of the address translation table