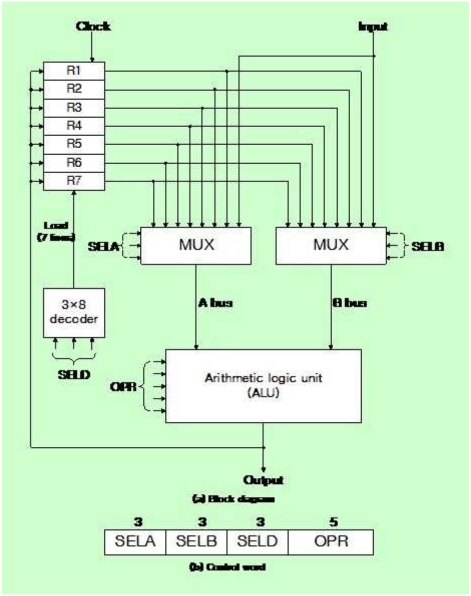
High performance CPU Architectures

Prof. K.R. Chowdhary, Director JIETCOE
Email: kr.chowdhary@jietjodhpur.ac.in

JIET COllege of Engineering

July 28, 2017

Typical Architecture: general register organization



Typical Architecture: general register organization

The number of registers in a processor unit may vary from just one processor register to as many as 64 registers or more. One of the CPU registers is called as an accumulator AC or 'A' register. It is the main operand register of the ALU.

Typical Architectures

- In computer science, computers a stack machine is a type of computer. In some cases, the term refers to a software scheme that simulates a stack machine.
- A stack computer is programmed with a reverse Polish notation instruction set.
- The common alternatives to stack machines are register machines, in which each instruction explicitly names specific registers for its operands and result.

Typical Architectures

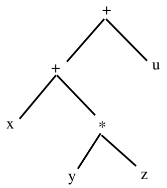
- For a typical instruction like “Add,” the computer takes both operands from the topmost (most recent) values of the stack.
- Memory is often accessed by separate Load or Store instructions containing a memory address or calculating the address from values in the stack.
- For speed, a stack machine often implements some part of its stack with registers. To execute quickly, operands of the arithmetic logic unit (ALU) may be the top two registers
- The instruction set carries out most ALU actions with postfix (Reverse Polish notation)
- In contrast, register machines hold temporary values in a small, fast array of registers.

Stack Machines Advantages

- Stack machines have much smaller instructions than the other styles of machines.
- In stack machine code, the most frequent instructions consist of just an opcode selecting the operation.
- In contrast, register machines require two or three register-number fields per ALU instruction to select operands;

Stack Machines: Simple compilers

- Compilers for stack machines are simpler and quicker to build than compilers for other machines.
- For example, given an expression x+y*z+u, the corresponding syntax tree would be:



```
push x
push y
push z
multiply
add
push u
add
```

- Such simple compilation can be done by the parsing pass. No register management is needed.
- This simplicity has allowed compilers to fit onto very small machines.

Stack Machines: Simple compilers

- The downside to the simplicity of compilers for stack machines, is that pure stack machines have fewer optimisations.
- Simple interpreters.
- Fast operand access.

Measuring & Reporting performance

- Response time: time between start and completion of an event.
- Response time is also called execution time
- Manager of a large data center may be keen in throughput- the total amount of work done in a given time.
- Let M_x and M_y are machines, and say M_y takes time t_y and M_x takes time t_x for some job.
- Let $\frac{t_y}{t_x} = n$, then M_x is n times faster than M_y.
- Let Performance of these machines are p_x and p_y, then

$$
\begin{aligned}
n = \frac{t_y}{t_x} = \frac{\frac{1}{p_y}}{\frac{1}{p_x}} \\
= \frac{p_x}{p_y}
\end{aligned}
$$

Measuring performance

- Execution related times: CPU time, user time, and system time.
- $time command runs programs and summarize system resource usage
- For example: $*time gcc prog*1*.c* produces output as:

```
$ time gcc prog1.c
  real 0m 0.309s
  user 0m 0.196s
  sys  0m 0.064s
```

- real: time by your watch, user: CPU time on account of user, sys: systems CPU time for IO, context switching, etc.
- System Performance refers to elapsed time on *unloaded system*, cpu performance refers to *user* cpu time on unloaded system
- We are interested on CPU performance.

Programs for measuring performance

- There are five levels of programs for measuring performance:
   1. Real applications: Real applications have input, output, and options. Applications are: word-processing, text editing, excel, photoshop, etc.
   2. Scripted Applications: Scripts are used to simulate applications.
   3. Kernels: Small key pieces are extracted and used as applications. Examples are "Livermore Loops", "Linpack."
   4. Toy benchmarks: 10 to 100 lines of code. Examples: "Sieve of Eratosthenes", "Puzzle", "Quicksort".
   5. Synthetic Benchmarks: Whetstone and Dhrystone are standard.

Programs for measuring performance

- One of the successful and standard benchmark is SPEC (Standard Performance Evaluation Corporation)
- www.spec.org
- Desktop benchmarks:
    1. These are two types: 1. CPU intensive benchmarks, 2) graphic intensive benchmarks.
    2. Other benchmarks are: transaction processing benchmarks, embedded benchmarks (in hardware)

Some Quantitative principles of Computer Design

- Make the common case fast: Favor the frequent over infrequent case. Improving the frequent case over infrequent will definitely improve the performance.
- Amdahl's law: We define speed up. Say, p_e is performance of the entire task using performance enhancement where possible, and p_n is performance with no enhancement done, then, we can define the,

$$speedup = \frac{p_e}{p_n} \tag{1}$$

Let t_e is execution time with enhancements, and t_n is time with no enhancement, then

$$speedup = \frac{t_n}{t_e} \tag{2}$$

Some Quantitative principles of Computer Design

- Let job $j_1$is 60 secs, and 20 secs of it can be enhanced, and rest cannot. So fraction enhanced is $20/60 = 0.33$. But, speedup is always > 1.
- Ideally, enhanceable can execute in 0 time (doing parallelism), so speedup is $60/40 = 1.5$.
- CPU Performance: Let P_c is cpu clock cycles for the program P, and T is clock cycle time. The CPU time for program:

$$t_{cpu} = P_c \times T \tag{3}$$

Some Quantitative principles of Computer Design

- If we know the total number of instructions in program P, say I_P, then we can find out “instructions per clock”. Inverse of this is “clocks per instructions” (CPI):

$$CPI = \frac{P_c}{I_P} \tag{4}$$

- Alternatively, from (3),

$$\begin{aligned} t_{cpu} &= I_P \times CPI \times T \\ &= \frac{I_P \times CPI}{f} \end{aligned} \tag{5}$$

Some Quantitative principles of Computer Design

- CPU Performance:
  1. Clock-cycle time (T) : depends on hardware technology and organization
  2. CPI: Organization and instruction set Architecture
  3. Instruction count (I_P): Instruction set architecture and compiler technology
- Some times it is useful to design CPU based on total cpu clock cycles (P_c) as:

$$P_C = \sum_{i=1}^{n} I_{P_i} \times CPI_i \tag{6}$$

where I_{P_i} represents the number of times the instruction I_{P_i} is executed in program P, and CPI_i is average number of clocks per Instruction i.

Classification of parallel Processors

- One classification by M.J. Flynn considers the organization of a computer system by the number of **instructions** and **data** items that can be manipulated simultaneously.
- The sequence of instructions read from the memory constitute an *instruction stream*.
- The operation performed on data in the processor constitutes a *data stream*.
- Parallel processing may occur in *instruction stream* stream or *data stream*, or both.

Classification of parallel Processors

1. Single-instruction single-data streams (SISD): Instructions are executed sequentially.
2. Single-instruction multiple-data streams (SIMD): All processors receive the same instruction from control unit, but operate in different sets of data.
3. Multiple-instruction single-data streams (MISD): It is of theoretical interest only as no practical organization can be constructed using this organization.
4. Multiple-instruction multiple-data streams (MIMD): Several programs can execute at the same time. Most multiprocessors come in this category.

Flynn's taxonomy of computer architecture(1966)

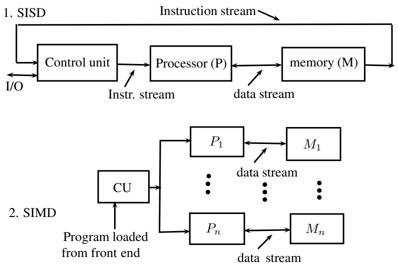


Figure 1: SISD and SIMD architecture

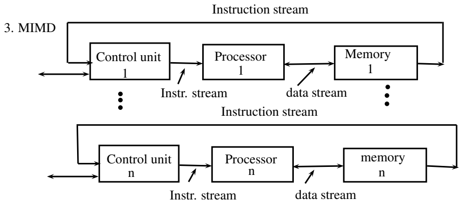


Figure 2: MIMD Architecture.

One of the parallel processing class that does not fit into this classification is *pipeline processing*.

Parallel processing

- Increasing speed by doing many things in parallel.
- If P sequential processor processes the task T in sequential manner, and if T is partitioned into n subtasks $T_1, T_2, \ldots, T_n$ of appox same size, then a processor P' having n processors can be programmed so that all the subtasks of T can execute in parallel. Now P' executes n times faster than P.
- Some applications of parallel processors:
  1. Fluid flow analysis,
  2. Seismic data analysis
  3. Long range weather forecasting,
  4. Computer Assisted tomography
  5. Visual image processing
  6. VLSI design
- A failure of a CPU is fatal to a sequential processor.
- Characteristic of parallel computing: vast amount of computation, floating point arithmetic, large number of operands.

Pipelining

- A typical example of parallel processing is a one-dimensional array of processors, where there are n identical processors $P_1 \ldots P_n$ and each having its local memory. These processors communicate by message passing (send - receive).

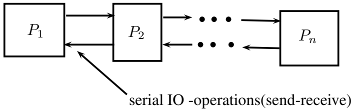
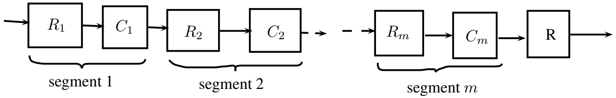


Figure 3: Pipeline processing.

- There are total n operations going on in parallel.
- A pipe line constitutes a sequence of processing circuits, called segments or stages.

- m stage pipeline has same throughput as m separate units.



Pipeline Processors

1. Instruction pipeline: Transfer of instructions through various stages of cpu, during instruction cycle: fetch, decode, execute. Thus, there can be three different instructions in different stages of execution: one getting fetched, previous of that is getting decoded, and previous to that is getting executed.
2. Arithmetic pipeline: The data is computed through different stages.

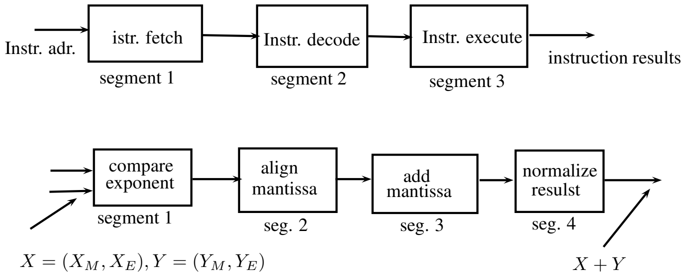


Figure 5: Instruction and data pipeline examples.

Pipe-lining Example

- Consider an example to compute: $A_i * B_i + C_i$, for $i = 1, 2, 3, 4, 5$. Each segment has r registers, a multiplier, and an adder unit.

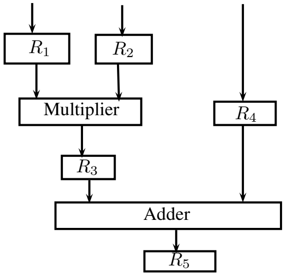


Figure 6: A segment comprising registers and computing elements.

$R_1 \leftarrow A_i$, $R_2 \leftarrow B_i$; input A_i, B_i
$R_3 \leftarrow R_1 * R_2$, $R_4 \leftarrow C$; multiply and i/p C
$R_5 \leftarrow R_3 + R_4$; add C_i to product

Pipe-lining Example

Table 1: Computation of expression $A_i * B_i + C_i$ in space and time in 3-stage pipeline.

Clock pulse no.	Segment 1 $R_1,\ R_2$	Segment 2 $R_3,\ R_4$	Segment 3 R_5
1.	$A_1,\ B_1$	$-,\ -$	-
2.	$A_2,\ B_2$	$A_1 * B_1,\ C_1$	-
3.	$A_3,\ B_3$	$A_2 * B_2,\ C_2$	$A_1 * B_1 + C_1$
4.	$A_4,\ B_4$	$A_3 * B_3,\ C_3$	$A_2 * B_2 + C_2$
5.	$A_5,\ B_5$	$A_4 * B_4,\ C_4$	$A_3 * B_3 + C_3$
6.	$- -$	$A_5 * B_5,\ C_5$	$A_4 * B_4 + C_4$
7.	$- -$	$-,\ -$	$A_5 * B_5 + C_5$

Pipe-lining Example

- Any operator that can be decomposed into a sequence of sub-operations can be implemented by pipeline processor.
- Consider that for a k-segment pipeline with clock cycle time $=t_p$ sec., with total n no. of tasks $(T_1, T_2, \dots, T_n)$ are required to be executed.
- T_1 requires time equal to $k.t_p$ secs. Remaining $n-1$ tasks emerge from the pipeline at the rate of one task per clock cycle, and they will be completed in time of $(n-1)t_p$ sec, so total clock cycles required $= k+(n-1)$.
- For $k=3$ segment and $n=5$ tasks it is $3+(5-1)=7$, as clear from table 1.

Pipe-lining processors

Instruction Pipe-lining: typical stages of pipeline are:

1. FI (fetch instruction)
2. DI (decode Instruction)
3. CO (calculate operands)
4. FO (fetch operands)
5. EI (execute instruction)
6. WO (write operands)

Instruction Pipe-lining

- Nine different instructions are to be executed
- The six stage pipeline can reduce the execution time for 9 instructions from 54 time units to 14 time units.

time units $\rightarrow$

Instruc.	1	2	3	4	5	6	7	8	9	10	11	12	13	14
In1	FI	DI	CO	FO	EI	WO								
In2		FI	DI	CO	FO	EI	WO							
In3			FI	DI	CO	FO	EI	WO						
In4				FI	DI	CO	FO	EI	WO					
In5					FI	DI	CO	FO	EI	WO				
In6						FI	DI	CO	FO	EI	WO			
In7							FI	DI	CO	FO	EI	WO		
In8								FI	DI	CO	FO	EI	WO	
In9									FI	DI	CO	FO	EI	WO

- The diagram assumes that each instruction goes through 6 stages of pipeline.
- But, for example, a load instruction does not need WO.
- It is also assumed that there is no memory conflicts, for example, FI, FO, WO in all require memory access (together).
- The value may be in cache, or FO/WO may be null.
- Six stages may not be of equal duration, conditional

Factors in Instruction Pipe-lining

- Overhead in each stage of pipeline for data movements buffer to buffer
- Amount of control logic needed to handle memory/register dependencies increases with size of pipeline
- It needs time for the buffers to operate
- Pipeline Hazard occur when pipeline or its portion stalls.
  1. Resource hazard: Two or more instructions in pipeline require same resource (say ALU/reg.) (called structure hazard)
  2. Data hazards: conflict in memory access
  3. Control hazards: (called branch hazards) wrong decision in branch prediction

Vector Processing

- In many computational applications, a problem can be formulated in terms of vectors and matrices. Processing these by a special computer is called vector processing.
- A vector is: $V = [V_1 V_2 V_3 \dots V_n]$. The index for V_i is represented as $V[i]$. A program for adding two vectors A and B of length 100, to produce vector C is:
- Scalar Concept:

```
for(i=0; i < 100; i++)
  c[i]=b[i]+a[i];
```

- In machine language we write it as:

```
      mvi i, 0
loop: read A[i]
      read B[i]
      store i = i +1
      cmp i, 100
      jnz loop
```

Vector Processing

- Accesses the arrays A and B, and only counter needs to updated. The vector processing computer eliminates the need of fetching the instructions, and executing them. As they are fetched only once only, decoded once only, but executes them 100 times. This allows operations to be specified only as:

$$C(1:100) = A(1:100) + B(1:100)$$

- Vector instructions includes the initial address of operands, length of vectors, and operands to be performed, all in one composition instruction. The addition is done with a pipelines floating pointing point adder. It is possible to design vector processor to store all operands in registers in advance.
- It can be applied in matrix multiplication, for $[l \times m] \times [m \times n]$.

RISC v/s CISC

CISC: (Complex instruction set computer)

- Complex programs have motivated the complex and powerful HLL. This produced *semantic gap* between HLL and machine languages, and required more effort in compiler constructions.
- The attempt to reduce the semantic gap/simplify compiler construction, motivated to make more powerful instruction sets. (CISC - Complex Instruction set computing)
- CISC provide better support for HLLs
- Lesser count of instructions in program, thus small size, thus lesser memory, and faster access

RISC v/s CISC

RISC: (Reduced instruction set computer)

- Large number of Gen. purpose registers, use of compiler technology to optimize register usage
- R-R operations (Adv.?)
- Simple addressing modes
- Limited and simple instruction set (one instruction per machine cycle)
- Advantage of simple instructions?
- Optimizing pipeline