Second Half of the Semester

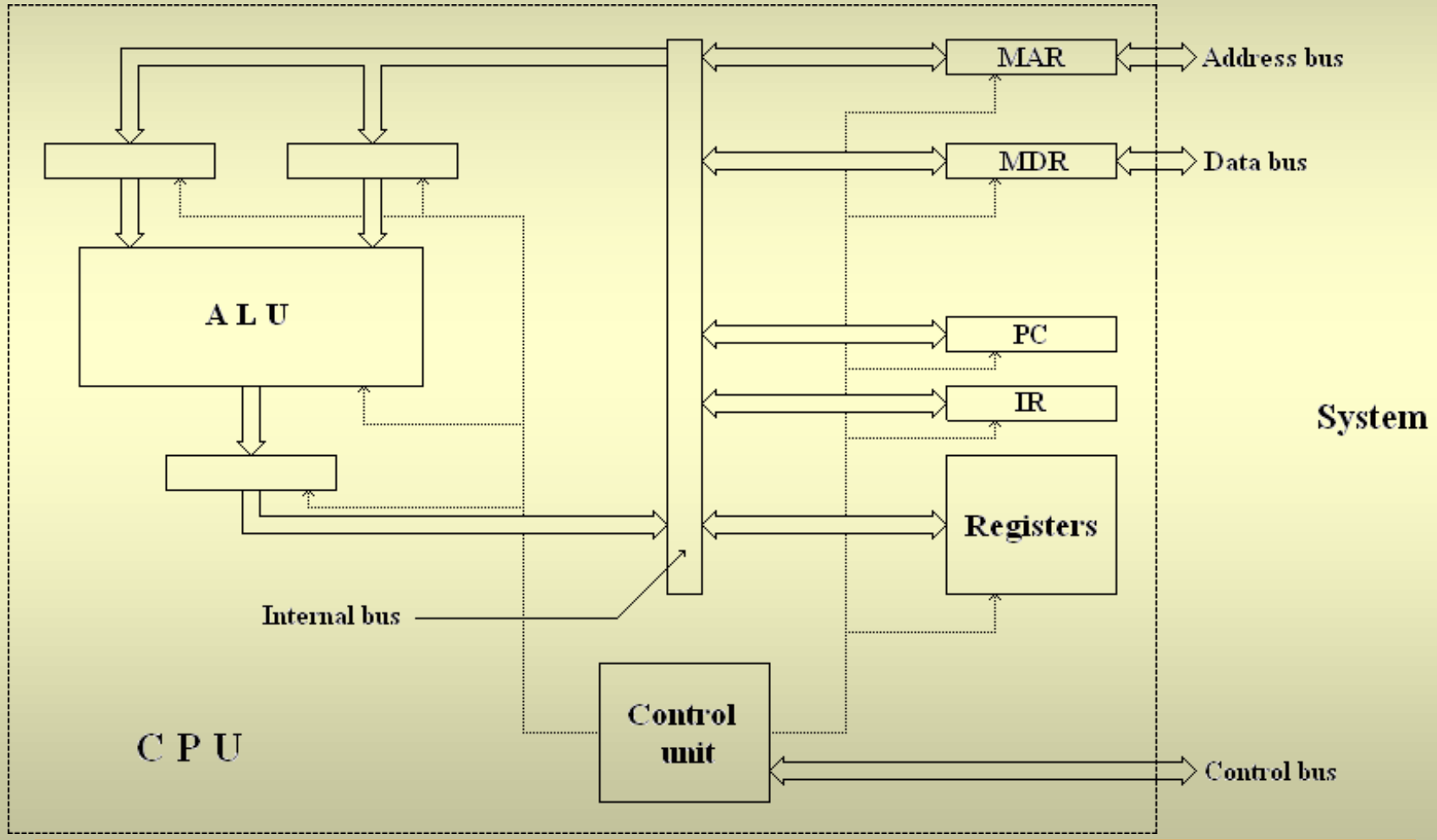
- course
  - the hardware structure of a computing system
  - evaluation - written test
    - reminder: in order to be accepted to the test - do not be absent more than twice at the laboratory classes
- laboratory
  - assembly language
  - evaluation - practical test

Course Contents

I. The Central Processing Unit (CPU)
II. Improving the CPU Performance
III. Peripheral Devices
IV. The Interrupt System
V. The Operating System

I. The Central Processing Unit (CPU)

CPU Structure (1)



CPU Structure (2)

- arithmetic logic unit (ALU)
  - performs calculations
- general-purpose registers
- control unit
  - commands the other components
  - sets up the time order of the operations
- internal bus

CPU Structure (3)

- program counter (PC)
  - keeps the address of the next instruction to be executed
  - updated by the processor
  - usually not accessible through the program
- instruction register (IR)
  - keeps the code of the last instruction fetched from memory

CPU Structure (4)

- interface registers
  - communication to the system buses
  - addresses: MAR (Memory Address Register)
  - data: MDR (Memory Data Register)
- temporary registers
  - intermediate between various components
  - examples: ALU operand registers, ALU result register

II. Improving the CPU Performance

How Can One Improve Performance?

- eliminate the factors that hinder the CPU
  - example - use of cache memory
- simple structures
  - no longer possible on today's processors
- increase clock frequency
  - limited by technological issues
- parallel execution of instructions

Improving the Performance - Techniques

- Pipeline structure
- Multiply the execution units
- Branch prediction
- Speculative execution
- Predication
- Out-of-order execution
- Register renaming
- Hyperthreading
- RISC architecture

II.1. Pipeline

Starting Idea

- executing an instruction - a lot of steps
- different CPU resources are used in different steps
- the execution of an instruction may start before the previous one was terminated
- instructions are executed (partially) in parallel

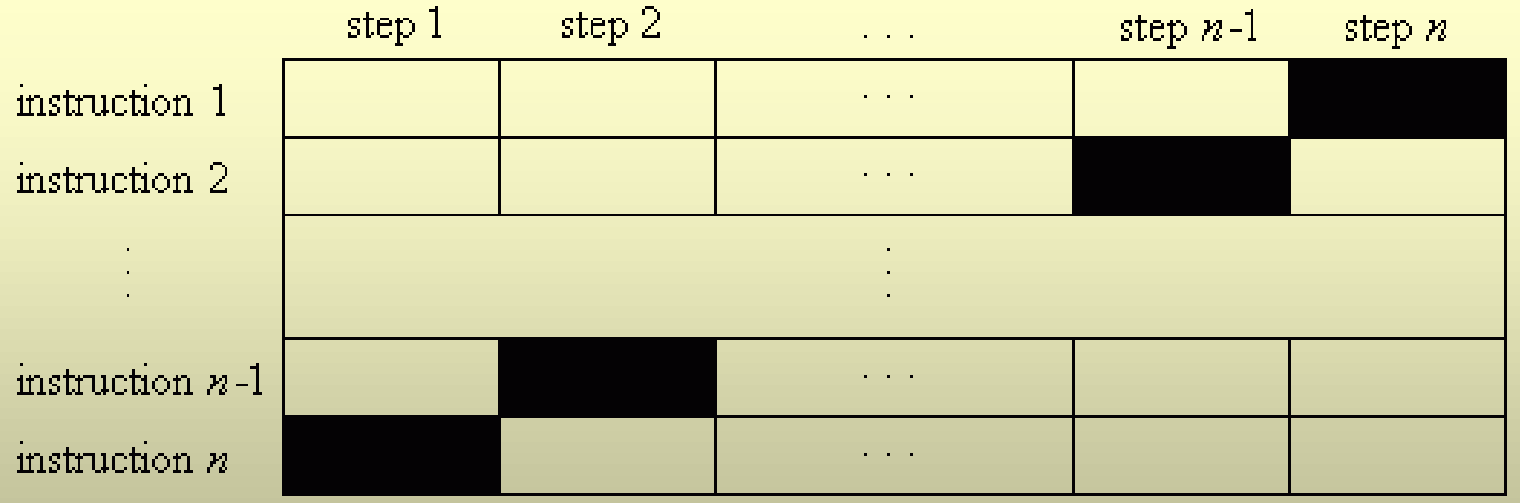
A First Implementation

Intel 8086 processor

- made of two units
  - bus interface unit (BIU)
    - communication to the outside
  - execution unit (EU)
    - proper execution of the operations
- BIU and EU can work in parallel

Assembly Line Principle

- execution of an instruction - *n* steps
- each moment - *n* instructions are executed
- each instruction - in a different step



Pipeline

- the sequence of steps (stages) the execution of an instruction goes through
- advance from one stage to the next - each clock cycle
- how long does it take until an instruction is terminated?
  - first instruction - *n* clock cycles
  - following instructions - 1 clock cycle each !

Pipeline Performance

- the result acquired on each stage must be kept
- separation registers - placed between stages
- clock frequency - given by the longest stage
- simpler steps
  - more stages
  - higher clock frequency

Instruction Execution

1. place the value of PC (instruction address) into MAR
2. issue the "read memory" command
3. get the instruction code into MDR
4. put the instruction code into IR
5. update the value of PC

Instruction Execution

6. the control unit decodes the instruction
7. read the operand from memory
   - place the address of the operand into MAR
   - issue the "read memory" command
   - get the instruction code into MDR

7'. select the register which contains the operand

Instruction Execution

8. put the operand into ALU's first operand register
9. repeat steps 7-8 for the second operand
10. send to ALU the code of the operation to be performed
11. put the result into ALU's result register
12. test jump condition

Instruction Execution

13. jump (if the previous test indicates that)
14. write the result into memory
    - put the result into MDR
    - put the address into MAR
    - issue the "write memory" command

14'. write the result into the destination register

Pipeline Evolution

- Intel Pentium III - 10 stages
- Intel Pentium IV (Willamette, Northwood) - 20 stages
- Intel Pentium IV (Prescott) - 32 stages

- AMD Athlon - 17 stages

Problems

- not all instructions can be executed in parallel
- dependency - an instruction must wait until another one is finished
- conflict for accessing the same resource

Performance Parameters

- *latency* - the number of clock cycles necessary for executing one instruction
  - given by the number of stages
- *throughput* - the number instruction finished per clock cycle
  - in theory - equal to 1
  - in practice - smaller (because of dependencies)

Dependency Types

- structural dependencies
- data dependencies
- control dependencies

Structural Dependencies

- instructions being in different stages need the same component
- only one instruction may use that component at a certain moment
- the other instructions that need it are blocked

Structural Dependencies - Examples

- ALU
  - arithmetic instructions
  - compute the operand addresses
  - update the value of PC
- memory access
  - read instruction code
  - read operand
  - write result

Data Dependencies

- one instruction computes a result, another one uses it
- the later instruction needs the result before the first one computes it
- the later instruction is blocked

Data Dependencies - Example

```
mov eax,7
sub eax,3
```

- first instruction: writing to `eax` - in the last stage
- second instruction: usage of `eax` - in the early stages (decoding)
  - waits until first instruction writes the result into `eax`

Control Dependencies (1)

Updating the value of the PC (usually)

- increase the current value with the size of the code of the previous instruction
- load a new value - jump instructions

Control Dependencies (2)

Types of jump instructions

- unconditional
  - always jump
- conditional
  - jump only if a certain condition is fulfilled
  - otherwise, continue with the following instruction

Control Dependencies (3)

Expressing the jump address

- a constant value
  - absolute
  - offset relative to the address of the current instruction
- a register value
- a memory location value

Control Dependencies (4)

Jump address - examples:

```
jmp 1594
jmp short -23
jmp eax
jmp dword ptr [esi]
```

Control Dependencies (5)

Problems

- computing the jump address - during the final execution stages
- the following instructions have already begun their execution
- if jump is taken - their effects must be cancelled

Control Dependencies (6)

Problems

- "emptying" the pipeline → loss of performance
  - complex operations
  - takes a long time until first instruction is finished → lower throughput
- about one instruction out of 7 is a jump !

Eliminating Dependencies

Solutions

- *stall*
- *forwarding*

Stall (1)

- occurs when an instruction needs a result which has not been computed yet
- that instruction "stalls" (does not proceed with next step)
- just like a "no-operation" (*nop*) instruction has been inserted
- we say a *bubble* has been inserted into the pipeline

Stall (2)

- the instruction moves on when the result it needs becomes available
- detection circuits are required
- it is not really a solution
  - does not truly eliminate the dependency
  - only guarantees the correct execution of the instructions
  - if an instruction stalls, the next ones will also stall

Forwarding (1)

```
add byte ptr [eax],5
sub ecx,[eax]
```

- addition result - computed by ALU
- takes time until written to destination
- the second instruction can take the addition result directly from the ALU

Forwarding (2)

Advantage

- reduces wait times

Drawbacks

- requires complex additional circuits
- must consider relations between all instructions being executed (into the pipeline)

II.2. Multiplying the Execution Units

Superscalar Units

- basic idea - multiple ALUs
- can perform more calculations in parallel
- used together with pipeline technique
- MAR and MDR cannot be multiplied
- how much can one multiply ALUs?
  - depends on pipeline's structure and efficiency

Superpipeline Units

- more pipelines inside the same processor
  - usually 2
- 2 (or more) instructions can be executed completely in parallel
- restrictions
  - memory and I/O accesses - always sequential
  - some instructions may be executed by the main pipeline only

II.3. Branch Prediction

Prediction (1)

- eliminate control dependencies
- basic idea - to "foresee" whether a jump will be taken or not
  - do not wait for the jump instruction to end
- accurate prediction - no stalls in the pipeline
- wrong prediction - execute instructions that should not have been executed
  - their effects must be "erased"

Prediction (2)

- performance improvement - more accurate predictions (not necessarily 100%)
- an instruction that should not have been executed only produces effects when the result is written to its destination
- instruction results - internally memorized by the processor until one can see whether the prediction was accurate or not

Prediction Schemes

Types of prediction schemes

- static
  - always make the same decision
- dynamic
  - adaptive, depending on program behavior

Static Prediction Schemes (1)

1. Jump is never taken

- accurate prediction rate ≈ 40%
- program loops
  - show often in the programs
  - jumps frequently taken

Static Prediction Schemes (2)

2. Jump is always taken

- accurate prediction rate ≈ 60%
- frequent errors - *if* structures

Static Prediction Schemes (3)

3. Jumps back are always taken, jumps forward are never taken

- combination of previous variants
- higher prediction rate

Dynamic Prediction Schemes (1)

- processor keeps the behavior on previous jumps
  - dedicated table (in hardware)
  - jump taken/not taken
- a single element for more jump instructions
  - smaller table → less space required

Dynamic Prediction Schemes (2)

Types of predictors

- local
  - keep information on single jumps
- global
  - consider correlations between jump instructions within the same program
- mixed

Intel Pentium

- *Branch Target Buffer* (BTB)
  - 4-way set-associative cache
  - 256 entries
- States of an entry
  - strong hit - take the jump
  - weak hit - take the jump
  - weak miss - do not take the jump
  - strong miss - do not take the jump

BTB Implementation (1)

State memorization and evolution

- 2-bit saturation counter
  - can count both up and down
  - value range - between 0 (00) and 3 (11)
  - from extreme states one cannot move further (only backwards)
- on each access, the state may change
  - jump condition is true - the counter increases
  - jump condition is false - the counter decreases

BTB Implementation (2)

- state coding
  - strong hit - 11
  - weak hit - 10
  - weak miss - 01
  - strong miss - 00
- why 4 states?
  - second chance - long term behavior

BTB Implementation (3)

current state	next state	
	jump condition true	jump condition false
00	01	00
01	10	00
10	11	01
11	11	10

Using the Cache for Prediction

Instruction cache

- keeps the previous behavior of a jump
  - condition
  - jump address
- *trace cache*
  - memorizes the instructions in the order they are executed
  - not in physical order