CS 320

Ch. 14 CPU Structure

The logic blocks inside the CPU are the **ALU Loop, Bus, Registers, Control Unit.**

The ALU does the following: **Shifting, Complementing, Arithmetic, Boolean operations, and keeps track of status.**

CPU registers may be user visible or control and status.
User visible means that they are available to the assembly language programmer.
Control and status are used for internal control and are not generally available directly to the assembly language programmer.

The user visible registers fall into four categories: General purpose; data, address, and condition codes.

A) General purpose registers: **Available for a variety of purposes. Can be orthogonal but this is not necessary or common.**
B) Data registers: **Registers used only to hold data and are not used in the calculation of an operand address. These may be part of the general purpose registers.**
C) Address registers: **Registers whose use is limited to holding addresses such as the index register, segment register, or stack pointer.**
D) Condition code registers: **Registers used to hold the status flag data.**

Some computer architectures such as the IA-64 do not use condition codes at all. **Conditional branches specify a comparison to be made and a branch based on the result without relying on storing the codes.**

The advantages of using conditions codes?
1. reduces number of compares
2. simplifies branches
3. can have multiway branches (>, ==, <).

The disadvantages of using condition codes:
1. add to complexity
2. condition codes are not regular and typically require specialized hardware to handle
3. require special synchronization in pipelined machines.

Register length is determined by the n**umber of bits in the op code and in the operands and the number of operands to be addressed. Operand size is dependent on the number of registers.**

Studies show that most computers need at least 8 registers and should have no more than 32.
Fewer than 8 results in too many memory references.

More than 32 don't get used very much and don't noticeably reduce memory references.

The four registers deemed essential for instruction execution which are classified as control or status are: **Program counter, instruction register, memory address register, and memory buffer register.**

The program status word is a **single register that holds all of the condition codes or flags.**

This figure shows the register sets of the Intel 8086 which has lots of special purpose registers and a compact instruction set with reduced flexibility. The Motorola 68000 has 8 data and 9 address registers, 8, 16, and 32 bit operands, no segmentation, and no special purpose registers.

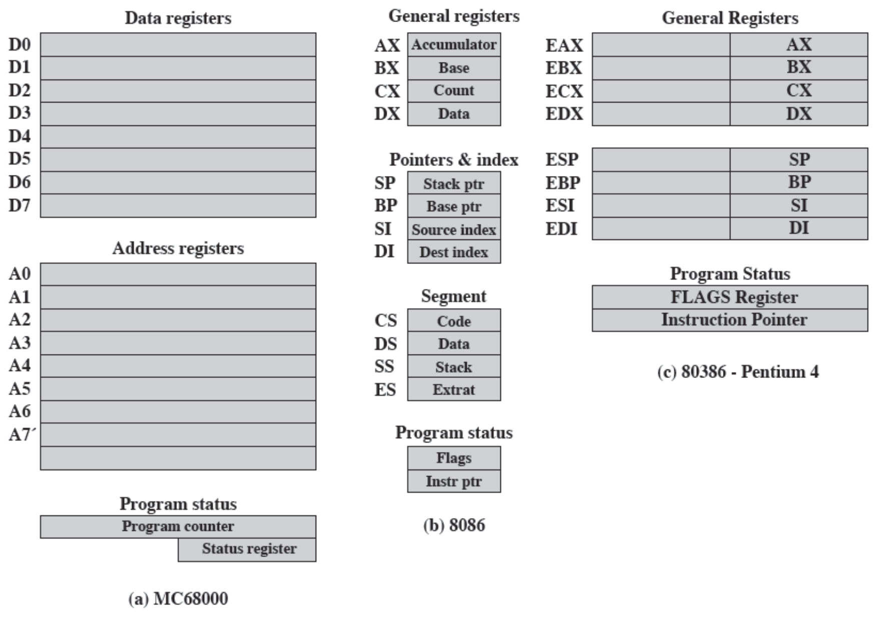


Figure 14.3 Example Microprocessor Register Organizations

The difference in design philosophy between the 8086 and the 68000 is:

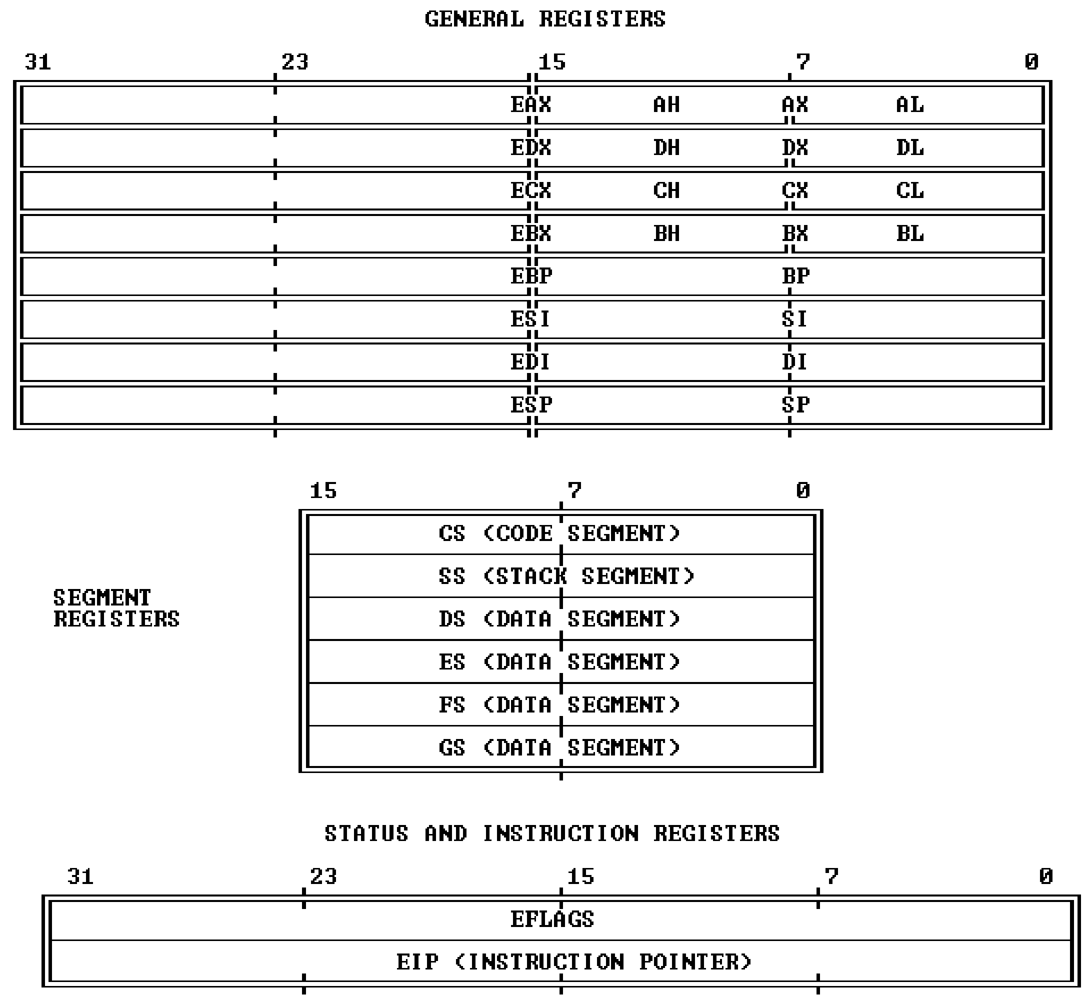
The **8086 has a specialized instruction set which allowed a more compact architecture. Hence it got to market nearly 2 years ahead of the 68000.**

The 68000 is more orthogonal and more general purpose in nature. It is, as a result easier to program although not necessarily faster or more efficient than the 8086.

The 68000 has 2 sets of 8 registers. One set is address and the other is data. This is a deviation from the general approach taken otherwise. This a**llows register address to be 3 bits instead of 4. For two operand instructions this saves 2-bits and shortens the instruction or allows more room for op codes.**

The 80386 register set is an extension of the 8086 register set. This allows it to be upward compatible to the older processor.

Figure 2-5. 80386 Applications Register Set



The instruction cycle may have 4 parts which are the fetch, execute, indirect cycle, and interrupt cycle. For the indirect cycle the **CPU goes back to memory with operand address and fetches data. Depending on whether this is register or memory indirect this may be more complicated.**

During the fetch cycle the RTL is:
PC → MAR
MAR→Address bus
Control unit does a memory read
M → MBR → IR.

The execute cycle differ from the fetch and indirect cycle in that the e**xecute cycle is unpredictable and takes many forms. Fetch and indirect cycle are regular and predictable.**

Instruction pipelining is a **way to do some parallel execution while maintaining a serial instruction stream. Similar to an assembly line.**

Pipelining does not speed up the execution of a single instruction. I**n fact in some cases it actually slows it down.**

Pipelining does speed up the throughput. **As in a car assembly line if you follow one car through the whole line it may take three days. On the other hand, if you watch cars come off the assembly line one car may come off as often as every 15 minutes.**

In a 2-stage pipeline the i**nstruction is broken up into a fetch and an execute. The two are overlapped.**

```
F  E
   F  E
      F  E
   . . .
```

Thus in an optimum case the execution time is doubled.

Note that a 2-stage pipe does not usually double the execution rate: **Execution time is typically longer than the fetch time and conditional branches and interrupts break up the flow. A pipeline can be clocked no faster than its slowest unit.**

There are 6 common pipeline stages:

Fetch instruction (FI)
Decode instruction (DI)
Calculate operands (CO) Find the effective address
Fetch operands (FO)
Execute instruction (EI)
Write operands (WO) store the result

Some reasons that an n-stage pipeline cannot run n times faster include:
Interrupts
Branches
FI, FO, and WO all use memory and you may not be able to overlap them in time
Instruction dependencies.

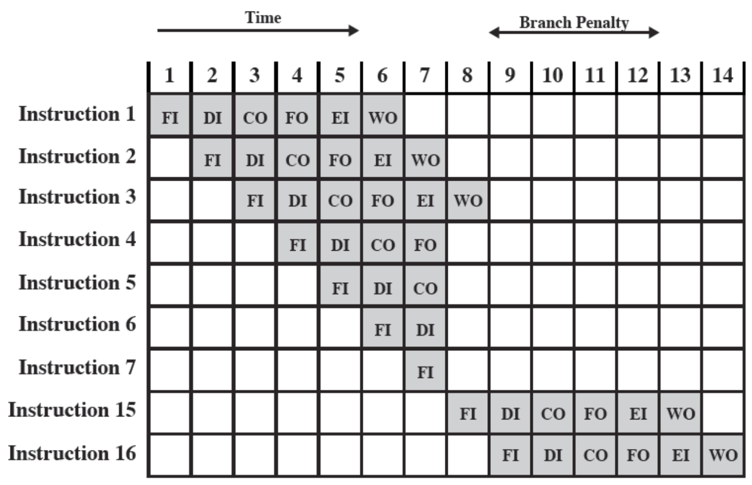
Assuming that all stages could be successfully overlapped, there are no interrupts, and there are no instruction dependencies, there is one more reason why an n-stage pipe couldn't be n times faster: **There is some overhead included in placing registers between stages. Also no stage can run faster than a stage which occurs later in time. In general all stages run at the same speed which is the speed of the slowest stage.**

Here is a simple example of instruction dependencies.

```
mov a, 5
mov b, 3
add a, b
mov M, a
sub m, 12
```

Can't do subtract of 12 from memory until previous move is complete.

28. Figure 14.11 p. 498 show a 6 stage pipelined machine:
   A) **Instruction 3 branches to instruction 15.**
   B) Instructions 4 through 7 are f**lushed.**
   C) The "branch penalty" in this case **caused a four time-period delay because we had to reload the pipeline**



	1	2	3	4	5	6	7	8	9	10	11	12	13	14
Instruction 1	FI	DI	CO	FO	EI	WO								
Instruction 2		FI	DI	CO	FO	EI	WO							
Instruction 3			FI	DI	CO	FO	EI	WO						
Instruction 4				FI	DI	CO	FO							
Instruction 5					FI	DI	CO							
Instruction 6						FI	DI							
Instruction 7							FI							
Instruction 15								FI	DI	CO	FO	EI	WO	
Instruction 16									FI	DI	CO	FO	EI	WO

Figure 14.11 The Effect of a Conditional Branch on Instruction Pipeline Operation

If you increase the number of stages of a pipeline, the author says that the control logic for the pipeline increases dramatically. **The more stages the greater the likelihood that one or more stages will cause an exception which will require reloading the pipeline. Also more stages means greater loss when the pipe is reloaded.**

A pipeline hazard is a **stall in the flow of the information through the pipeline that prevents further execution.**

A pipeline hazard may be caused by a l**ack of resources, data conflict, or branches.**

The author lists three classifications for pipeline hazards:
Resource hazard – two or more instructions in the pipe need the same resource.
Data hazard – two or more instructions need the same data.
Control hazard – also called a branch hazard.

The author lists three types of data hazards.

Read after Write (RAW) – also called true data dependency. One instruction writes data and a following instruction reads it. A hazard occurs if the read comes before the write.
Write after Read (WAR) – Also called an anti-dependency. One instruction reads a register and a following instruction writes it. A hazard occurs if the write completes before the read.
Write after Write (WAW) – also called an output dependency. Two instructions write to the same location. A hazard occurs if the writes happen in the wrong order.

The author lists 5 ways to deal with branches:

Multiple streams
Prefetch branch target
Loop buffer – buffer big enough to contain one loop
Branch prediction
Delayed branch

multiple streams: **Create two pipelines and execute both sides of a branch in parallel. When the condition is decided discard the wrong result.**

prefetch branch target: **Fetches address of both sides of a branch.**

loop buffer: **This is effectively a small cache which holds all of the instructions in the loop. This is in addition to a normal system cache.**

Branch prediction: **Use some algorithm (possibly instruction stream history) to predict whether or not a branch will be taken. Execute the prediction ahead of time.**

The author lists five common techniques that are used to do branch prediction.
Predict never taken
Predict always taken
Predict by opcode
Use a taken/not taken switch
Create a branch history table.

Branch prediction may be static and dynamic:
Static does not depend on execution of current instructions and dynamic does. For example, one static strategy might be to predict the branch taken for all jnz instructions.
A dynamic strategy might be to predict a branch if taken if it was taken the last time.

delayed branch: **Assume that the instruction immediately following any branch will always be executed since it is already in the pipe. Rearrange the instructions so that this following instruction does something useful. For example,**

```
mov a, 5
dcr c
jnz target
add d,b
```

In this case the add d, b instruction is wasted if it is in the pipe and the jnz is taken. We can rearrange this set of instructions to look like this:

```
dcr c
jnz target
mov a, 5
add d,b
```

The *move a, 5* instruction will always be executed since it is in the pipe but this time it is doing something useful since it was supposed to always have been executed.

The 80486 has a 5-stage pipeline:

Fetch – independent stage tries to keep a 16-byte buffer full. This hold, on average about 5 instruction.
Decode stage 1 – basic decoding of op code and addressing modes.
Decode stage 2 – op code expanded into ALU control signals and decodes more complex addressing modes
Execute – ALU ops and register updates
Write back – updates registers and status flags. Also writes to memory if necessary.

The effective throughput of the 486 pipeline is a**bout 1 instruction per cycle.**

If one stage of the 486 pipeline is not needed – say the D2 stage **A nop is inserted.**

In the 486 pipeline if a conditional jump is taken the p**ipeline is flushed and new instruction is started.**

Complex instructions introduce new problems into pipelines: **Delays may be introduced or the pipeline can temporarily stall while an instruction is resolved. Also there is considerable hardware overhead for complex instructions and especially for complex addressing modes.**

For now we will skip the ARM architecture

CS 320 **Name __________________**
In Class **March 14, 2018**
Pipelines Ch. 14

1. Assume a pipeline has four stages: fetch instruction (FI); decode instruction (DI); fetch operand (FO); and execute (EX). Draw a diagram for a sequence of 7 instructions in which the third instruction is a conditional branch to instruction 15 which is taken. Assume there are no instruction dependencies.

	1	2	3	4	5	6	7	8	9	10	11
I1											
I2											
I3											
I4											
I5											
I6											
I7											
I15											

2. A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has 5 stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instruction and out-of-order sequence executions.

A) What is the speedup of this processor for this program compared to a one pipelined processor?

B) What is the throughput, in MIPS, of the pipelined processor?

CS 320 **Name _ANSWERS__**
In Class **March 14, 2018**
Pipelines Ch. 14

1. Assume a pipeline has four stages: fetch instruction (FI); decode instruction (DI); fetch operand (FO); and execute (EX). Draw a diagram for a sequence of 7 instructions in which the third instruction is a conditional branch to instruction 15 which is taken. Assume there are no instruction dependencies.

	1	2	3	4	5	6	7	8	9	10	11
I1	**FI**	**DI**	**FO**	**EX**							
I2		**FI**	**DI**	**FO**	**EX**						
I3			**FI**	**DI**	**FO**	**EX**					
I4				**FI**	**DI**	**FO**					
I5					**FI**	**DI**					
I6						**FI**					
I7											
I15							**FI**	**DI**	**FO**	**EX**	

2. A pipelined processor has a clock rate of 2.5 GHz and executes a program with 1.5 million instructions. The pipeline has 5 stages, and instructions are issued at a rate of one per clock cycle. Ignore penalties due to branch instruction and out-of-order sequence executions.

A) What is the speedup of this processor for this program compared to a onpipelined processor?

We can ignore the initial filling up of the pipeline and the final emptying of the pipeline, because this involves only a few instructions out of 1.5 million instructions. Therefore the speedup is a factor of five.

B) What is the throughput, in MIPS, of the pipelined processor?

One instruction is completed per clock cycle, for an throughput of 2500 MIPS