

Master's Thesis:

The design and implementation of a pipelined 8051 microcontroller core

By R.A.M.J. Snijders

Coach : Dr. Ir. A.C. Verschueren
Supervisor : Prof. Ir. M.P.J. Stevens
Period : November 1996 – October 1997

Abstract

This master's thesis describes the pipelining of the 8051 microcontroller core. The reason for this assignment is the CAN controller, which is being developed at the section of Information and Communication Systems. This CAN controller consists of three processors of which one is the application processor. This application processor will be a fast 8051 based microcontroller core. The speed makes it necessary to use the pipelining technique.

The idea of pipelining is to split up the basic function of the microcontroller in more subfunctions. For each subfunction a hardware module is designed, which is called a stage. These stages can operate independent of each other and form a pipeline through which a continuous instruction stream flows. Inter-instruction dependencies prevent the microcontroller core from running at its optimal speed.

The functionality of the 8051 core is split up in the following stages:

- Instruction Addressing
- Instruction Receiving
- Instruction Decoding
- Operand Addressing
- Operand Receiving
- Instruction Execution
- Operand Write Back

Necessary mechanisms are designed to handle the inter-instruction dependencies, reduce the amount of used hardware and increase the performance. Techniques like Early Branch Calculation and Data Forwarding are used

The design is implemented and tested and can run at a clock frequency up to 40MHz. The performance of the pipelined 8051 is approximately a factor 10 better than the standard running at 25 MHz. Some extensions, reductions and improvements of the system are suggested.

Contents

1	Introduction	1
2	Pipelining the 8051 microcontroller core	3
2.1	The standard 8051 core	3
2.2	Pipelining in general	4
2.3	Pipelining applied to the 8051	5
3	The microcontroller core	7
3.1	The environment of the core	7
3.2	The core divided in stages	8
4	Dependencies	10
4.1	Dependencies in general	10
4.2	Structural dependencies in the application	11
4.3	Data dependencies in the application	12
4.4	Stalling of the pipeline	14
5	Branching (Program flow change)	16
5.1	Branching in general	16
5.2	Solving branch problems in the application	18
6	Instruction fetch	19
6.1	Accessing ROM	19
6.2	Program Counter Handling	20
6.3	Arranging instruction bytes	20
7	Instruction decoding	22
7.1	Address calculations	22
7.2	R0 and R1 Shadow Registers	23
7.3	Source and destination information generation	24
7.4	Resolving instructions in this stage	25
7.5	Solving the hazards	25
8	Operand fetching	27
8.1	Operand addressing	27
8.2	Operand receiving	27

8.3	Accessing memory	28
9	Instruction execution	29
9.1	The ALU	29
9.2	Testing conditional branches	30
10	Memory access	32
11	System Extension, reduction and performance	33
11.1	System extension	33
11.2	System reduction	34
11.3	System performance	35
12	Conclusions and recommendations	37
12.1	Conclusions	37
12.2	Recommendations	37
	References	38
Appendix 1	Standard instruction set	40
Appendix 2	Standard memory map	41
Appendix 3	Data and address flow in the pipeline	42
Appendix 4	Control flow in the pipeline	43
Appendix 5	Source and destination information format	44
Appendix 6	Penalty table	46

1 Introduction

CAN (Controller Area Network) is a network used in the automotive industries. It was developed in the 70's and nowadays it is supported by various companies like Philips, Intel and Siemens who all include CAN controllers in their line of products. Most of these controllers have an integrated microcontroller used to run the network protocol. Also using this microcontroller to run an application program will probably be too much.

At the technological University of Eindhoven, Faculty of Electrotechnical Engineering, the Information and Communication Systems department is developing a CAN controller consisting of two processors handling the network protocol and one integrated microcontroller core which is totally available for running the application. Communication between the two processors and the microcontroller core is controlled by the interface module. Figure 1.1 gives a simplified impression of the CAN controller. More about CAN and the CAN controller can be read in [HOFM96].

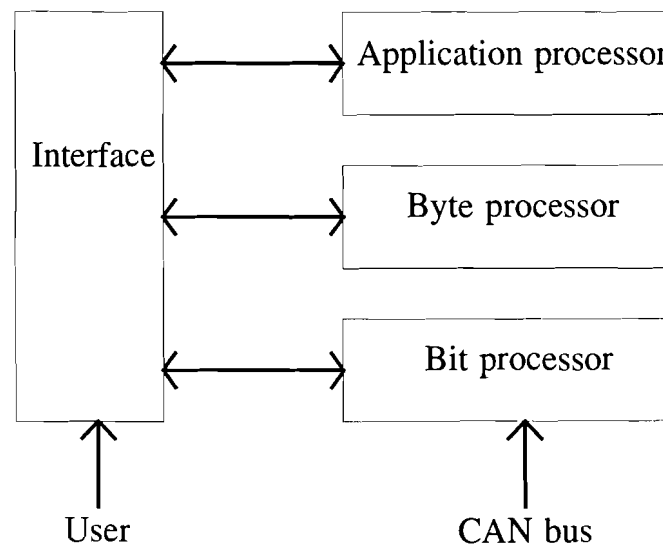


Fig. 1.1 Simplified overview of the CAN controller

In [HOFM96] it was decided that this integrated microcontroller should be an Intel 8051 core. The reason for this choice is the widespread use of this microcontroller and the great amounts of software already written for it. To be suitable for being used in the CAN controller some adjustments to the standard 8051 have to be made. These adjustments mainly concern the memory map and the system performance. To increase the speed of the 8051 it will be designed using pipeline techniques.

Pipelining is a technique to improve system performance by pursuing instruction-level parallelism. The technique is nearly as old as electronic computers, but became very popular

in the last few years because it is a cheap way of improving system performance, especially since the limits of the used technologies are in sight.

This report will describe the functional design and implementation of an Intel 8051 compatible microcontroller core using pipelining techniques. The problems that arise using this technique will be handled and possible solutions will be presented. Also a motivation of the used solutions is given. The implementation and testing is done in IDaSS [VERS90].

2 Pipelining the 8051 microcontroller core

In this chapter first a brief overview of the standard 8051 core and the pipelining technique is given. After this I will describe the problems which arise by applying pipelining on the 8051.

2.1 The standard 8051 core

The standard 8051 core is described best by its memory map, instruction set and addressing modes. An overview of the instruction set is given in appendix 1.

The 8051 uses separate program and data memory. An overview is given in appendix 2. The program memory can be divided in internal and external memory. The first two kilobyte can be either internal or external. Selection is done via an external pin. The higher addresses are always external.

Data memory can also be internal or external. The internal block consists of the four Register Banks, bit addressable RAM, the higher part of internal RAM and the Special Function Registers. Some of the Special Function Registers are also bit addressable. The difference in accessing internal or external ram is made by using different addressing modes.

The 8051 uses five addressing modes:

Register Addressing accesses the eight working registers (R0-R7) of the selected bank. Bank selection is done via two selection bits in the Program Status Word. The Accumulator, B register, Data Pointer and Carry can also be addressed as register.

только младших байт; старших байт адреса "0"
Direct Addressing is the only method of accessing the Special function Registers. The first 128 bytes of RAM is also directly addressable.

адрес задан в регистре (??)
Register-Indirect Addressing uses the contents of either R0 or R1 of the selected register bank as a pointer to the internal RAM or the first 256 bytes of external RAM. Access to the full external RAM is accomplished by using the Data Pointer. The PUSH and POP instruction also use Register-Indirect Addressing.

Immediate Addressing allows constants to be part of the instruction in Program Memory. *↑ байт значения сразу за командой.*

Base-Register Plus Index Register-Indirect Addressing allows a byte to be accessed from Program Memory via an indirect move from the location whose address is the sum of the Program Counter or Data Pointer and the Accumulator.

More information about the 8051 can be found in [INTE83].

2.2 Pipelining in general

During the mid 1960's the price of hardware dropped sufficiently to allow computer architects to replicate computer subsystems as a technique for developing high performance systems. This replication of hardware allows different pieces of hardware to handle distinctly different parts of the problem. This is the technique we call pipelining.

The idea of pipelining is to split up the functions in more subfunctions. In [KOGG81] there are given the following properties of the partitioning.

- 1 Evaluation of the basic function is equivalent to some sequential evaluation of the subfunctions.
- 2 The inputs for one subfunction come totally from outputs of previous subfunctions in the evaluation sequence.
- 3 Other than the exchange of inputs and outputs, there are no interrelationships between subfunctions.
- 4 Hardware may be developed to execute each subfunction.
- 5 The times required for these hardware units to perform their individual evaluations are usually approximately equal.

The hardware used to evaluate such a subfunction is called a stage. This hardware normally doesn't contain any memory. Between the stages registers are included to keep a discrete and synchronised data flow. The data at the input of these registers should be stable at the end of a clock cycle. For these registers various names are used like staging registers, staging platforms, reservation station or steady points. In this text I will use staging registers or just registers.

The increase in performance is obtained by letting different instructions use different stages at the same time. This means that a stage should be able to perform its subfunction independent from the activities of other stages. Now more instructions can be handled at the same time. If normally an instruction would take 6 clock cycles and we were going to execute 50 of those instructions it would take 300 clock cycles. Using the pipelining technique with a pipeline of six stages it will only take 56 cycles (50 functions plus 6 to fill the pipeline). In figure 2.1 this is visualised for 5 instructions using a reservation table.

Time →	1	2	3	4	5	6	7	8	9	10	11
Stage 1	1	2	3	4	5						
Stage 2		1	2	3	4	5					
Stage 3			1	2	3	4	5				
Stage 4				1	2	3	4	5			
Stage 5					1	2	3	4	5		
Stage 6						1	2	3	4	5	

Fig. 2.1 reservation table of stages

Unfortunately the pipelining technique almost never gives a performance as shown in figure 2.1. There are some problems which influence the program flow through the processor. These problems can be divided in two categories. **Branching Problems** or Program Flow Changes occur when a Jump instruction, a Call or a Return is executed. In case of a conditional Jump the evaluation is done at the end of the pipeline. The preceding part of the pipeline should be filled with an instruction flow depending on the outcome of the evaluation. Some techniques are available to solve this problem. More details about the problem and its solutions will be given in chapter five.

Hazards or dependencies occur due to the design or the use of the pipeline so that the pipeline is prevented from accepting data at the maximum rate that its staging clock might support. Hazards occur independent of the program flow. There are two categories of hazards: Structural and data dependent. A structural hazard happens if two instructions attempt to use the same stage at the same time. For obvious reason these are also called collisions. Data dependent hazards happen if two instructions in different stages attempt to use the same data. More details about this problem will be given in chapter four.

Another problem about pipelining is the additional hardware needed, and with this the additional costs. In order to let the pipeline perform like it should be a lot of mechanisms are needed to control data flow, access of memory and the use of shared hardware. This makes the design of a pipeline more like finding a compromise between performance and costs.

With pipelines some classification is possible according to [KOGG81]. If a pipeline is capable of evaluating only one kind of function it is called a *unifunction pipeline*. If it is possible to evaluate more kinds of functions it is called a *multifunction pipeline*. If a multifunction pipeline is only capable of infrequent changes of the function being run then it is *statically configured*. If a multifunction pipeline is capable of more frequent changes up to a different function each input, it is called *dynamically configured*. The time an instruction uses the facilities offered by a stage is called *latency*. Generally this latency can be any integer including 0.

Pipelining is a technique classified under the general term of concurrent operation. Other techniques are parallelism and overlap, of which the last one is a primitive form of pipelining. I only name these techniques not to get any confusion. I won't describe the differences because it is out of the scope of this paper. More about this subject can be read in [KOGG81].

2.3 Pipelining applied to the 8051

The department of Information and Communication Systems wants to have a fast microcontroller at their disposal. There is a need for this controller in the 'CAN controller' project, but also for other applications such a controller could come out handy.

The controller will be Intel 8051 based. This means that the instruction set and the memory map correspond to the standard 8051. For now only the core will be designed at a functional level. It will be used embedded in other designs. The core should be running at the speed of

approximately 33MHz. This can be achieved by using pipeline technique described in the previous paragraph. It should be designed as cheap as possible, so the controller architecture will be a compromise between performance and cost. The designing and testing will be done in the environment of IDaSS.

Problems to be expected are mainly staging, solving hazards and controlling branches. For this, solutions have to be found according to the performance/cost compromise. Mechanisms have to be designed to increase the performance of the controller. Possible mechanisms are data forwarding and the use of shadow registers, both to decrease the amount of memory accesses. Also mechanisms have to be designed to decrease the cost of hardware. Here we can think about arbiter mechanisms to share hardware.

3 The microcontroller core

In the previous chapter I told that the controller should be 8051 based. In the CAN controller the environment of the 8051 core is created by the interface described in [SLEG96]. This interface makes it possible for the three processors to work simultaneously. This environment, offered to the core is not totally conform the standard of the 8051. Therefore, in this chapter some adjustments to the memory map will be made. In this chapter I also will describe the staging, the first step towards a pipelined architecture.

3.1 The environment of the core

The environment of the controller is provided by the Interface of the CAN controller. There are three different types of memory to be accessed. **ROM** contains the program code. It can be up to 64 Kbyte of size. It is divided into an internal and an external block, but this is not perceptible by the microcontroller core. The ROM is shared with the byte processor of the CAN controller witch has a higher priority. This is also not perceptible by the microcontroller. The priority of access, and access to either internal or external program memory is regulated by the interface. Because of sharing this memory a delay can arise in the delivery of the data. This is solved by using a data-ready signal. This signal is explained later in this paragraph.

The **RAM** in the CAN controller is organized not in the same way as in the standard. In the CAN controller there is no difference between internal or external RAM in the lower address range. In this range all attempts to access RAM are projected on the internal block. This internal block has a size of 512 bytes and contains the four banks and the bit addressable space on the same addresses as in the standard. The remainder of the 64 kbytes are external. Also the RAM is shared. Again the priority of access and the access to either internal or external RAM is not perceptible by the microcontroller and regulated by the interface. Here also a data-ready signal is used which will be described later in this chapter.

The **SFR** space (Special Function Register) is also different as in the standard. A lot of Special Function Registers are not present because the function they are used for is not implemented. The only Special Function Registers used by the core are the Accu, B register, Data Pointer, Program Status Word and the Stack Pointer. The Stack Pointer is extended to 16 bits. No sharing is done here so the data can be expected immediately and no ready signal is used.

The 8051 core will have a separate program, data and SFR bus. Summarized, the microcontroller core has got the following interface with the environment:

ROM:	Address (16 bits), Data in (8 bits), Read (1 bit), Ready (1 bit).
RAM:	Address (16 bits), Data in (8 bits), Read (1 bit), Data out (8 bits), Write (1 bit), Ready (1 bit).
SFR:	Address (7 bits), Data (8 bits), Write (1 bit), Read (1 bit).

The ready signal, used by accessing RAM or ROM is implemented in a special way to speed up operation. The microcontroller gets an early ready when reading. If the microcontroller initiates a read in cycle n , it will get the ready signal in cycle n (under normal conditions). This makes it possible for the microcontroller to start a next read in cycle $n+1$. The data becomes valid in cycle $n+1$. This is all under the assumption that we use synchronous memory. A normal read operation would need $2 \cdot M$ cycles for M accesses. Using the technique described above only needs $M+1$ cycles. When writing, a ready signal is given in cycle n under normal operation.

3.2 The core divided in stages

The function of the microcontroller core is to execute the program stored in ROM. This means executing all possible instructions sequentially according to the program flow. We have to design a dynamically configured multifunction pipeline.

The first step in pipelining is dividing the function into subfunctions. With a microcontroller it means that we have to find a partition which suits all instructions. A possible way is:

- Instruction fetch
- Operand fetch
- Instruction Execution
- Write back operand

The addressing of the memory is done in two steps, addressing and receiving data. Before operands can be fetched their addresses should be calculated. In order to calculate those addresses the instruction should be partially decoded. A more suitable partitioning is the following:

- ROM addressing
- Receiving instruction bytes
- Decoding instruction
- Addressing operand memory (RAM, ROM, SFR)
- Receiving operand
- Actual instruction execution
- Write back operand

This fits for every instruction if we keep in mind that an instruction should be able not to use the facilities offered by a certain stage. Such an instruction is only supposed to wait in that stage one cycle. In the above partitioning I also assumed that each stage has got a latency of 1. This means that all instructions should stay in a certain stage at least one cycle (also if they only wait) and all the activities done in a stage should be finished within one cycle. The only exception on this will be the execution stage in case of a multiply or divide instruction and the stage in which the instruction bytes are collected.

According to the partitioning of the function of the microcontroller in more subfunctions, the pipeline can be divided into stages. The stages with their functionality are the following:

Stage 0, ROM Module. This stage addresses ROM. In the remainder of this theses it will be referred to as the ROM module

Stage 1, Instruction Fetch. This stage collects the bytes from the ROM module to form a complete instruction, and passes this instruction to the next stage.

Stage 2, Instruction Decoding. In this stage the instruction is partially decoded. With this information the source and destination addresses of the operands will be calculated. This stage also decides whether the register used to calculate the addresses is valid. In order to do these calculations a 16-bit ALU is present. In this stage the Data Pointer and Stack Pointer are kept.

Stage 3, Operand Addressing. The addressing and the verification whether the addressing is permitted, are done in this stage. No instruction decoding or address calculation is necessary because this is all done in the previous stage.

Stage 4, Operand Receive. The operand is received and a kind of data forwarding is done to update the R0 and R1 Shadow Registers.

Stage 5, Instruction Execution. The actual instruction execution is performed in this stage. The ACCU, B register and the Program Status Word are kept here. Two busses between this stage and stage 2 make it possible for the registers kept over there to be used as source or destination. This stage also performs the conditional jump testing.

Stage 6, Write Back. Operands are written back and a bus between this stage and stage 3 makes data forwarding possible. A bus to stage 2 is added for keeping the Ri Shadow Registers up to date.

Beside these stages for each memory type a module is added. These mainly take care of the priorities used within the core, so the stages are not bothered with it. I also have given the Program Counter a separate module because of the handling of program flow changes. For this it needs to be controlled by stage 1, stage 2 and stage 5.

In the system two kinds of information flows are present. The data flow is needed to let the instruction flow from one stage to another or to forward data. A diagram is given in appendix 3. Everything will be explained in the following chapters. In order to control the data flow, stages must be able to control other stages and modules. This is done by control signals. A diagram of the control signals is given in appendix 4 and will also be explained in the following chapters. A detailed description of the microcontroller core can be found in the documentation of the design.

4 Dependencies

One of the biggest problems in a pipelined design is the handling of dependencies. I will use this chapter to explain the problem in general, show where this problem occurs in the application and give an idea how to solve it.

4.1 Dependencies in general

In case of a non-pipelined design all the operations involved in executing a single instruction are finished before the next one is started. This is not true in a pipelined design. Here instruction executions are overlapped and it is possible that operations involved in executing an instruction are not finished when operations involved in executing succeeding instructions already started. This gives problems if those operations are dependent of each other. Those dependencies cause hazards that must be detected by hardware, and resolved. There are two different kinds of hazards. The **structural hazard** occurs if two instructions tend to use the same piece of hardware at the same time.

The **data dependent hazard** occurs whenever a data object (register, memory location, flag) is accessed by different instructions of which the execution overlaps. There are three classes in which this kind of hazard can be divided: *read-after-write* (RAW) hazard, *write-after-read* (WAR) hazard and *write-after-write* (WAW) hazard. There is also a read-after-read dependency but this never causes a hazard. I will explain the three hazards using an example.

Let's assume the following instruction sequence:

MOV	\$34, #\$16	"write value \$16 into memory location \$34"
INC	\$34	"read memory location \$34, increment value, write value back to memory location \$34"
MOV	\$34, #\$18	"write value \$18 to memory location \$34"
MOV	\$34, #\$19	"write value \$19 to memory location \$34"

A RAW hazard exists if an instruction wants to read but the previous instruction still has to write to the same data object. In the above program fragment this happens between the first move and the increment instruction. If the increment instruction reads location \$34 before the move instruction has written, it will be incrementing the wrong value.

A WAR hazard exists if an instruction wants to write before the previous instruction has read to the same data object. In the example this is the case between the increment and the second move instruction. If the move instruction writes the value to memory before the increment instruction is able to read, it will also be incrementing the wrong value.

A WAW hazard will exist if two instructions write to the same data object in the wrong order. In the example this is the case between the increment and the second move instruction.

If the increment instruction writes later to memory than the move instruction, the memory location will contain the wrong value afterwards.

In a pipeline, which executes the instructions in the order in which they are taken out of memory and which only has got one stage for reading and one later stage for writing, the WAW hazard and the WAR hazard won't happen if the data objects are memory locations. If the data objects are registers kept by one stage in the pipeline all hazards are still possible.

4.2 Structural dependencies in the application

A structural dependent hazard exists if two instructions try to use the same piece of hardware. In the application four pieces of hardware are shared, so at four places we can expect problems

The first three shared hardware blocks are the memory access points. In fact, the only shared part is the address bus. Different types of memory is accessed by the following stages:

- Stage 1: ROM read, to receive instructions from ROM.
- Stage 3: ROM read, to receive operands.
RAM read, to receive operands.
SFR read, to receive operands.
- Stage 6: RAM write, to store operand.
SFR write, to store operand.

For each type of memory there are two stages able of accessing it. These stages might want to access it at the same time. Priority must be given to one of the stages. This has got to be the stage later in the pipeline in order not to make it unstable. I will explain this with an example:

Imagine a situation in which stage 3 and stage 6 both want to access RAM and stage 3 has got priority. Also imagine stage 6 being able to restrain stage 3 from performing its function (This is caused by stalling the pipeline, explained in a later chapter). Both stages access RAM but stage 6 is stopped, based on priority. This causes stage 6 to stall the pipeline and indirectly restrain stage three from accessing memory. The request for accessing RAM by stage 3 is abolished, so stage 6 can access RAM. Stage 6 can proceed and won't stall the pipeline anymore. Stage 3 is not stopped because there is no stall and wishes to access RAM. Stage 6 is prevented to access RAM and will stall the pipeline again.

For each type of memory there is need for some kind of arbiter. I solved this problem by using a ROM-, RAM- and SFR-controller, handling the priority.

The fourth part of shared hardware is the 16-bit ALU of stage 2. This ALU is used to calculate operand addresses. To save hardware, only a 4-bit incrementor is used, present in the Program Counter Module, for incrementing the Program counter. Once every 16 times the Program Counter is incremented by the 16-bit ALU of stage 2. For this, the Program Counter Module

has got to place a request at stage 2. Again we get the problem of priority. If we would give priority to the Program counter module, the system might get unstable. But if we would give the priority to stage 2 the first part of the pipeline might run empty.

A pipelined system can only get a maximal performance if the incoming instruction flow is as high as possible. Because of this it is preferable to give priority to incrementing the Program Counter. The problem with the instability is solved by buffering the incoming program flow and only requesting an increase by stage 2 if it is really necessary (if an access of ROM in request of stage 1 is done).

4.3 Data dependencies in the application

There are three kinds of hazards caused by data dependencies between instructions. The data objects that they relate to are the following:

RAM locations. Reading is done by stage 3 and writing by stage 6.

Special Function Registers not kept in the core. Reading is done by stage 3 and writing by stage 6.

Special Function Registers kept in the core. These are the ACCU, B register, DPTR, SP and the PSW.

In case of memory locations we can simplify the problem. The core will be executing the instructions in order and reading memory will always be done in an earlier stage than writing. Because of this WAW and WAR hazards are not possible. We only have to solve the RAW hazards.

The only possible way to detect these dependencies is to compare the addresses of the different instructions in the pipeline. Writing will be done in stage 6, Reading in stage 3, so we have to compare the source address of stage 3 with all the destination addresses of the following stages. If there is a match, no reading can be done and the instruction of stage three should wait till the dependent instruction has floated out of the pipeline. If we imagine two dependent instructions in stage three and stage 4, it will cause at least a delay of three clock cycles. It is possible to reduce this delay using data forwarding.

Data forwarding is a technique to improve performance in a pipelined system. As the name already says, data will be pushed back into the pipeline. In this implementation I will use a bus to forward data from stage 6 to stage 3. If the two depending instructions will be in stage 3 and stage 6, a possible data forwarding is detected and the instruction in stage 3 can go on using the data from stage 6. This will, on first sight, give an improvement of 1 cycle. But it also cancels the need to access memory, and with this any chance of a further delay. This is visualised in figure 4.1 b.

Time →	1	2	3	4+i	5+i+j	6+i+j	7+i+j	8+i+j
Stage 1								
Stage 2								
Stage 3	A	A	A	A				
Stage 4	B				A			
Stage 5		B				A		
Stage 6			B				A	

a. No Data Forwarding

Time →	1	2	3	4	5	6	7	8
Stage 1								
Stage 2								
Stage 3	A	A	A					
Stage 4	B			A				
Stage 5		B			A			
Stage 6			B			A		

b. Single Data Forwarding

Time →	1	2	3	4	5	6	7	8
Stage 1								
Stage 2								
Stage 3	A							
Stage 4	B	A						
Stage 5		B	A					
Stage 6			B	A				

c. Multiple Data Forwarding

Fig. 4.1 Data forwarding

An improvement of the Data Forwarding Mechanism can be made if data is also forwarded to stage 4 and stage 5. With this improvement the delay is reduced to zero.

Data objects that are more difficult to handle are the registers kept in the core. They are used by stage 2 and stage 5. In these stages these registers have the following purposes:

Stage 2	DPTR	Address calculation, destination register.
	SP	Address calculation, destination register.
	ACCU	Address calculation.
	B	Not used.
	PSW	Indicates which bank is used, so indirectly it is used for address calculation.

Stage 5 All registers are used as source or destination register.

The B register will not give any problems. The ACCU and PSW may cause RAW hazards. The DPTR and SP may cause RAW hazards, WAW hazards and WAR hazards. We can solve this problem by stopping stage 2 every time a dependency is detected. For this we need to know from every instruction what their destinations and sources are. In stage 2 I will add to every instruction a source and destination word containing all necessary information. From this information it can be decided which register can be used in stage 2.

4.4 Stalling of the pipeline

In every stage it might happen that an instruction stays for more than one clock cycle. This can be caused by a data dependent hazard, structural hazard or any other reason an instruction should need the facilities of a certain stage more than one cycle. If this happens, the stages earlier in the pipeline should be informed about this so they can hold the instruction they are handling in order not to get any collisions. In fact, informing a previous stage about congestion is preventing a structural hazard.

The technique to prevent structural hazards in case of any congestion is called *stalling*. The signal used to inform a previous stage about this situation is called a *stall*. The stalling of a system can be done centralised or decentralised. Centralised means that there would be a controller managing the information of the whole system. Decentralised gives the responsibility of local stalling to each stage.

In this implementation I have chosen for a decentralised solution. A stage only needs to be stalled if the stage in front of it is occupied. If a stage is stalled, it gives a stall to the stage behind it. The whole system of stalling is in fact a chain of local stalls. A stall in the last stage can cause an indirect stall in the first stage. This type of stall mechanism is called a ripple stall. A simplified representation is given in figure 4.2.

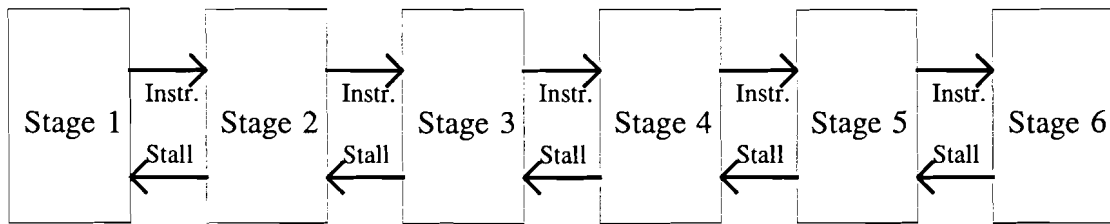


Fig. 4.2 Ripple stall

A problem with a ripple stall is the time that the stall signal needs to pass through the system. Especially if the stall signal is dependent of other signals which are stable late in the clock cycle this will be a problem. A possible solution is buffering the pipeline at critical points. In the application a critical point is stage 3 where the advanced ready signals from RAM and ROM influence the stall signal. These advanced ready signals have the property to be stable late in the clock cycle.

5 Branching (Program flow change)

5.1 Branching in general

Control-flow-instructions refer to branch, call and return instructions. In this text I will refer to them as branches. These instructions are used to change the sequence of a program. To execute a branch a processor must perform two operations:

- Compute the target address
- Evaluate the condition that determines whether a branch should be taken

The latter is only necessary in case of a conditional branch.

The most used kind of branch is the *Program Counter Relative branch*. Here the target is a displacement relative to the current position of the branch instruction. In some other cases the target address must be computed using memory locations or registers. This type is called *computed branch* and includes certain jump and return instructions. There are also some branch instructions which include their target address in the instruction code.

Conditional branch instructions specify a condition when the branch should be taken and, therefore, whether the next instruction should be taken from the target address or the next sequential address. Unconditional branches can be seen as a particular subset of which the condition is always true. In this text I assume that the test done for branching is included in the branch instruction code.

When executing a branch, the following hazards may occur:

- 1 The instruction that is to be executed after the branch depends on the result of the branch condition.
- 2 The operands to evaluate the branch condition depend on the result of earlier instructions.
- 3 The execution of the next instruction can not be started until the target address is calculated.
- 4 The operands to calculate the target address depend on the result of earlier instructions.

Due to the frequent use of branches, these hazards might cause a substantial reduction of processor performance. It is of great importance to solve them in such way that the time lost is reduced to a minimum.

There are, according to [GONZ94], five basic ways to reduce the cost of branches. In the remainder of this paragraph I will describe them shortly to give an impression of the possibilities.

Delayed branch is one of the most widely used techniques. This is due to the fact that it doesn't require any hardware support. The delayed branch technique is based on redefining the branches in such way that it implies a disruption on the sequence of a program several instructions later and not on the place where it appears. Delaying the effect of a branch removes the dependency with the instruction just placed below. The distance between the branch and the depending instruction prevents this dependency from causing a hazard. The same happens for the dependency between the branch instruction and the instruction at the target address. The created distance should be filled by the assembler or the programmer with independent instructions.

Early computation of branches is a technique in which the place where the branch takes effect is moved forwards in the pipeline. Normally the two operations involved in handling a branch are performed by the ALU. Performing these operations in the beginning of the pipeline will increase performance. Another way to advance target computation is using a Branch Target Cache. This memory is accessed parallel with the fetch of the branch instruction and, in case of a hit, the processor has the target address available just after the branch has been fetched. Another possibility of using a cache is using it as Branch Target Instruction Cache. In this case not the target address is advanced but the instruction at the target address. A combination of these two is also possible.

Branch prediction advances the branch condition by predicting it. Obviously the processor must be able to cancel the instructions already started in case of a wrong prediction. Such a prediction can be done statically or dynamically. In the first case the prediction is done by the compiler. This means that the instruction code must be able to contain this information. Dynamic predictions are carried out each time a branch is executed. In order to do this the behaviour of the branch instruction in previous executions should be known. This behaviour should be stored in some way.

Branch bypass eliminates the need for predicting the branch condition. It just executes both paths and discards one of them when the outcome of the branch condition comes available. For this technique some hardware of the processor must be duplicated. It is possible that such a path also contains a branch instruction and therefore both paths should also be split up again, and so on. A particular case of this technique is the case in which there is only one branch and both possible paths are just fetched. This technique is called **Multiple prefetch** and only requires an increase in memory accesses.

Parallel execution of branches overlaps the executions of branches totally with the executions of the instructions. For this technique the instruction unit should be totally uncoupled from the execution unit and connected via a cache memory. The instruction unit handles the fetching of instructions and executing of the branch instructions. The instructions are stored in a cache with the address of the following instruction. The execution unit runs the program from this cache. In case of a conditional branch the instruction is stored in cache with two addresses and the execution unit uses a prediction technique. This technique can be seen as an extension of the Multiple prefetch technique.

5.2 Solving branch problems in the application

When solving branch problems we have to keep the following in mind. The controller core is supposed to be based on the 8051 standard. This includes that it should be able to run already written software in machine code assembled by already existing assemblers. So, the technique used to minimise branch penalties should entirely be implemented in hardware. The hardware used should be limited to a minimum. So, the use of additional memories should be avoided.

Because of this the Delayed Branch technique, Branch Prediction and Parallel Execution of branches are not possible to use. They have to be implemented in software, need duplication of hardware blocks or need cache memories. A simplified form of Branch Bypassing in which the second path is not prefetched, is difficult to implement concerning the flushing of the pipeline. If a branch is taken the already partly executed instructions have to be taken back and the changes they made must be corrected. This involves additional hardware and a much more complicated design.

For now only Early Computation of the branch target address will be implemented. This means that the computation of the target address must be placed in the beginning of the pipeline. In stage 2 we will need a 16-bit ALU to calculate the operand addresses. We will also use this ALU to calculate the branch target address. Sharing the ALU won't give any complications because none of the branch instructions needs operand address calculation. In case of the `JMP @A+DPTR` we have to reckon with the validity of the ACCU and the Data Pointer.

Early Computation of the branch condition is not moved to the beginning of the pipeline. The test is often made on a data object which has got to be taken out of memory or which is kept in the ALU stage (stage 5) and is influenced by the instructions right in front of the branch. The computation of the condition therefore can not be done earlier than in the ALU stage. Of course unconditional branches can be handled totally in the second stage.

Before an instruction is recognised as a branch it has got to be fetched from memory and partially decoded. This will take 2, 3 or 4 clock cycles, depending of the number of bytes of which the instruction consists. Loading the Program Counter with the right value takes 1 more cycle in case of an unconditional branch and 4 more cycles in case of a conditional jump (if not stalled by previous instructions). Altogether a branch will cause a delay of 3, 4 or 5 clock cycles in case of an unconditional branch and 7 or 8 cycles in case of a conditional branch (all conditional branches consist of 2 or 3 bytes).

6 Instruction fetch

In order to run a program, instruction bytes have to be fetched from program memory (ROM). The address at which they are fetched is the program counter. These instruction bytes are put together to form instructions who are shifted into the pipeline. In order to save hardware, some instructions are replaced by sequences of other instructions. All this is done by the ROM Module, Stage 1, and the Program Counter Module.

6.1 Accessing ROM

The actual addressing of ROM is done by the ROM module. For this it only needs an address which is the Program Counter and a Read Signal. This Read Signal is controlled by several signals coming from the stages.

From Stage 1 it is controlled by the Stall Signal. This Stall Signal, being a ripple Stall, has got the property to get active late in the clock cycle. The addressing should be done as early in the cycle as possible. To overcome this problem there are two possible solutions. The first is to break off the access later in the clock cycle and ignore any data received. This solution makes it necessary to start addressing again if the Stall is deactivated and gives some problems in keeping the Program Counter up to date. The second, a more clean solution, is to make the Read Signal not directly dependent of the Stall Signal. This can be done by buffering the input from ROM.

In the implementation we have chosen to finish already started accesses. In this way the time spent in addressing is also used optimally. In order to realise this, a two stage FIFO is used. A read can be done if there is free space in this FIFO. If a Stall arises the buffer can be filled in order to speed up performance just after the Stall. Notice that a one-register buffer does not make the Read Signal independent from the Stall. If a byte comes from ROM and a Stall arises, the byte is put into the buffer. Before the Stall arose, in the same clock cycle, a Read was already started. Because the Stall can still be active the next cycle, the Read has got to be broken of. This makes it still dependent of the Stall Signal

In case of branches and Program Counter dependent addressing the increased Program Counter value is needed to calculate target or operand addresses. For this, the ROM module sends with each instruction byte the Increased Program Counter Value (pointing at the next instruction) towards Stage 1. This means that in case of buffering the instruction byte, also the increased Program Counter value should be buffered. Because no addressing will be done if the second level of the FIFO is used, the PC will not be increased. The PC will maintain its value and therefore shouldn't need a second level of buffering. In figure 6.1 the buffering is visualised.

Another signal controlling the ROM module is the PFC (Program Flow Change). This signal indicates that the accesses done are (possibly) not in the right program flow because of a

branch, which is going to take place. If this signal gets active the stored information in the FIFO is thrown away and the started addressing is broken of.

The ROM module offers the instruction bytes to stage 1 with an accompanying ready signal. Towards the program counter module a signal is given to increase the program counter every time an advanced ready from ROM is received.

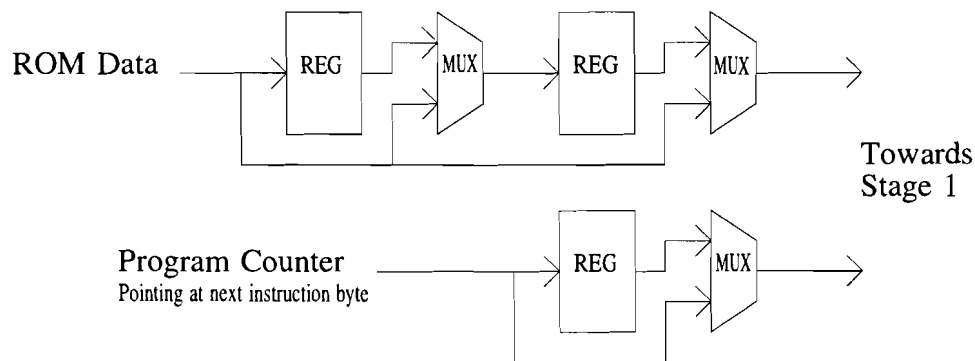


Fig. 6.1 Receiving of instruction bytes

6.2 Program Counter handling

To keep the program counter up to date, every time an addressing is done it is increased. Every time a branch instruction is executed, it should be updated with the right value.

Incrementing is simply done with an incrementor. It should be 16 bits wide and therefore quite expensive. To keep the costs of hardware low, the Program Counter Module only makes use of a 4-bit incrementor. Every 16 increments, the program counter is once incremented by the 16-bit ALU of stage 2. For this the Program Counter Module has got to place a request at Stage 2 and load the incremented value from Stage 2 into the PC register.

In case of branches the Program Counter should be loaded with the target address or, in case of conditional branches where the branch is not taken, the Program Counter should hold the original value (pointing at the next instruction if the branch wouldn't be taken). Holding this original value may cause some problems. The buffering and the type of Ready Signal given by ROM make it quite complex to be sure about this original value. In order to avoid these problems and to simplify the design, the Program counter is loaded with the Program Counter Value kept with the branch instruction. In case of a branch taken, the Program Counter is loaded with the target address under control of stage 2 in case of unconditional branches and under control of stage 5 in case of conditional branches.

6.3 Arranging instruction bytes

Stage 1 receives instruction bytes from the ROM module. Such an instruction byte can be the

first, second or third byte of an instruction. To put the bytes on the right place, stage 2 makes use of a Finite State Machine. According to the first byte received, the State Machine places the other bytes (if there are other bytes) in the right registers. If all the bytes of an instruction are present the instruction is offered to stage 2.

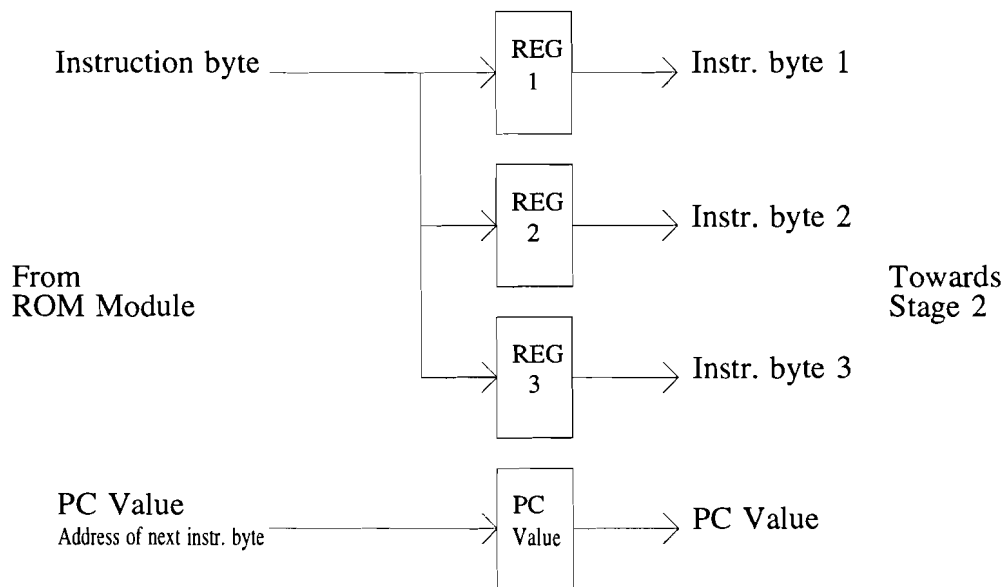


Fig. 6.2 Arranging instruction bytes

The activities of Stage 1 can be stalled by means of a Stall Signal coming from Stage 2. If this signal becomes active Stage 1 collects the bytes of the present instruction and when the instruction is complete it stalls the ROM module.

In order to save hardware costs and to simplify the design of the next stages, some instructions are replaced with a sequence of other instructions. For example, a Call is replaced by a Jump and two Pushes of the Program Counter (high and low byte) and a Return is replaced by two Pops towards the Program Counter. This makes it necessary to implement the Program Counter as a Special Function Register.

To simplify the design of the next stage the sequence of the instruction bytes is not maintained. This means that with some instructions the second or third byte is not put in the second respectively third instruction register. In case of an absolute jump, the instruction byte is copied into the second register to simplify target address calculation.

7 Instruction decoding

In stage 2 the source and destination addresses are generated. First the instruction is partly decoded in order to decide what kind of addressing should be used. Sometimes the address is not given directly in the instruction code. This means that a lot of information has got to be retrieved and calculated. This will be described in paragraph 6.1. From this information the source and destination addresses are composed and it should be checked if the registers used for composing these addresses are valid.

7.1 Address calculations

The standard 8051 supports 5 addressing modes: Register, Direct, Register Indirect, Immediate and Base Register plus Index-Register Indirect addressing. The actual memory can only be accessed with a direct address. In this paragraph the necessary calculations are described for each addressing mode.

Register Addressing is used to access the eight working registers in the active bank. The last three bits of the instruction code indicate which register should be used. With these three bits it is easy to calculate a direct address, which can be used for addressing. According to the standard it is also possible to access the ACCU, B register, DPTR and CY with this addressing mode. These registers are all kept within the core. So, calculating an address is not useful.

Direct Addressing is used to access SFR space and the lower 128 bytes of RAM. If the direct address is above address 128, it is meant for accessing SFR (If most significant bit is one). The address is already included in one of the instruction bytes. When accessing SFR space it is necessary to check if the address corresponds with a Special Function Register kept within the core. If so, no addressing should be done and the information should be included in the source or destination information shifted towards Stage 3, which is needed to solve hazards.

In case of a bit addressed operand, a direct address must be calculated to get the byte (which contains the bit) from memory. The memory is not bit addressable, only byte addressable. A mask must be derived to retrieve the bit from the received byte. The mask must be added to the instruction information, which will be sent to the next stage. The mask will only be used in the execution stage (Stage 5). If in the execution stage the bit is manipulated, it has got to be put back in the byte, which will be written back to memory. So, in case of using bit addressing as a destination in memory, always a direct source and destination address is necessary.

Register-Indirect Addressing uses working register R0 or R1 from the active bank as a pointer to the lower 256 bytes of RAM. To speed up the performance of the system, the value of those registers is kept in Shadow Registers present in this stage. This prevents the core

from repeatedly accessing these registers in RAM every time an Indirect Addressing is done. More about these Shadow registers will be written in the next paragraph. The SP and the DPTR are also used for indirect addressing and therefore they are kept in this stage.

Immediate Addressing allows constants to be part of the instruction in program memory. There is no addressing involved, so no address needs to be calculated.

Base-Register plus Index Register-Indirect Addressing uses a base register and an offset to indicate the address. This base can be either the DPTR or the PC and the offset will always be given by the ACCU.

For the last addressing mode an addition should be done and this creates the need in this stage to perform more complex calculations. These calculations are done in the 16-bit ALU. The functions it should be able to perform are moving a 16-bit value, incrementing SP and DPTR, decrementing SP and adding an 8-bit value to a 16-bit value. In order to save hardware and speed up branches, this ALU will also be used to calculate relative and absolute target addresses. Sharing the ALU this way gives no conflicts because instructions will never need both operand and target address calculation. The ALU can be built up of an 8-bit adder and an 8-bit incrementor/decrementor. This is sufficient to perform its operations.

7.2 R0 and R1 Shadow Registers

If we use the registers R1 or R0 for register indirect addressing, first the contents of these registers has got to be fetched from memory. This will give a delay of 2 clock cycles. In order to decrease this delay, Shadow Registers within the core are used, which contain the actual value of the real R0 and R1 registers in memory. If register indirect addressing based upon R0 or R1 is used, the active bank must be known. For this the PSW should be stable and the registers should not be written to.

There are four pairs of Ri registers (i stands for 0 or 1). But the only ones used, are those in the active bank. The value of these registers must be available in the shadow registers. There are two ways of implementing these shadow registers. We can use a shadow register for each possible Ri register. This would take eight registers. Those eight registers would always be up to date because we can easily track every access made to the originals. In practise a bank will be active for a longer time and switching between banks will only be done sometimes. This means that always only two of the eight registers are used.

A cheaper solution is only to use two shadow registers, only for the Ri registers of the active bank. It will save the cost of six registers but also creates the need to keep the shadow registers up to date, and this can get quite tricky because of the bank switching.

The Shadow Registers will become invalid if bank switching is done. Bank switching can be done by changing the Bank Bits in the PSW. In practise, almost every time there is written towards the PSW, it is done to change the Bank Bits. So, we can assume that a write action towards the PSW makes the Shadow registers invalid.

Updating can be done by reading the registers from RAM and putting the received value in the Shadow Registers. Reading is done by forcing an instruction towards the next stage. In this implementation there is chosen to force a NOP with an address. We must be able to detect the receipt of data from R0 or R1 in stage 4. This data must then be forwarded to stage 2.

Accessing RAM for updating the Shadow Registers can be done every time the Shadow Registers are invalid and Register Indirect Addressing is needed. It is also very well possible to change the Bank and immediately initialise R0 or R1. For this it is handy also to detect a write towards R0 or R1 in stage 6 and forward the data towards stage 2.

Imagine the case in which we don't change the Bank Bit when writing to the PSW or when we switch bank to store something and switch back again. In this case it is not necessary to update the Shadow Register. With an extension of the described mechanism it is possible to handle this scenario.

When updating the Shadow registers we also store the number of the active bank. Now imagine the following situations:

PSW is valid. If a read or write is detected towards R0 or R1 of the active bank, the Shadow Registers are updated together with the number of the active bank.

PSW is invalid and a read or write is detected towards R0 or R1 in the Bank corresponding with to stored Bank Number. In this case we can't speak of an active bank. In order to keep the Shadow Registers up to data when a switch is done back to the bank corresponding to the stored bank number, the register value is updated but not the bank number. If the PSW becomes valid and the stored Bank number is corresponding to the active Bank, the Shadow Registers will be up to date.

PSW is invalid and a read or write is detected towards R0 or R1 in a bank not corresponding to the stored Bank Number. No updating is done. It is not known to which bank there will be switched so updating the Shadow Registers is useless. Maintaining the old value is as good as shadowing a register from another bank.

This technique needs information from stage 4 and stage 6 about a possible detection and the number of the addressed bank. This information consists of the register value, an indication if this value is R0, R1 and a Bank Number, which indicates the bank of the detected R0 or R1 register. Of course, also information about the validity of the PSW is needed.

7.3 Source and Destination information generation

Together with the instruction code, information about the source and destination must be shifted towards stage 3. This information is needed to address memory and detect dependencies to solve hazards. The 'source word' and the 'destination word' must contain the following information:

Address. This always should be a direct address. It is calculated in the 16-bit ALU or

retrieved in some other way.

Type of memory to be addressed. This can be ROM (not in case of destination), RAM or SFR-space. This information is present in the instruction code and therefore some instruction decoding must be done. In case of Direct Addressing, this information can be extracted out of the address.

Long or short addressing. This information can also be extracted from the instruction code by means of instruction decoding.

Register information. If the addressing mode is direct, it can be towards a Special Function Register kept in the core. This should be detected and included in the source or destination information. Whether the ACCU is used as source or destination can be extracted from the instruction code. In case of a branch instruction the PC is used as destination.

All this is put together in a format, given in appendix 5. In case of generating the destination addressing, there also should be checked if this destination is not R0 or R1. If it is, the format should be adjusted and the according bits should be set.

7.4 Resolving instructions in this stage

To speed up performance and to simplify the design of the following stages it is always useful to finish instructions as soon as possible. In this stage such instruction resolving can be done to a number of instructions. They include instructions concerning registers kept in this stage and unconditional branches:

NOP resolves in every stage and is generated in every stage in case of a stall.

AJMP, LJMP, SJMP, JMP @A+DPTR. The target address is calculated and the Program Counter is updated. No more activities are necessary to solve the instruction.

MOV DPTR, INC DPTR. These instructions handle the DPTR and this can be done in this stage under the condition that the DPTR is not used as source or destination in an earlier instruction in the pipeline.

7.5 Solving the hazards

In order to solve the hazards first the dependencies must be detected. Such dependencies can cause a hazard and if so, they have got to be dealt with. First the possible dependencies are given:

The next stage gives a stall and the present instruction cannot resolve here. If the instruction will resolve, there would not be any problem.

In calculating the operand address, registers are used which are not valid. The value in the used registers can be incorrect because the register is used as a destination in an earlier instruction. Also notice the fact that the PSW should be valid in case of Register addressing (R0..R7) and Register indirect addressing (R0 or R1).

Using a register as destination, which is also the source of an earlier instruction. This is the case with a move or increment of the DPTR or SP. Notice that the SP is incremented or decremented in case of a PUSH respectively POP instruction.

The address used as destination possibly is pointing at R0 or R1 and the PSW is not valid. In order to be sure, we have to wait till the PSW becomes valid. It is necessary to have this information to detect dependencies with succeeding instructions.

The only way of solving these hazards is to wait. This is done by giving a Stall Signal to stage 1. If stage 3 gives a Stall Signal the stage registers between stage 2 and stage 3 should hold their value, if not, a NOP should be forced towards stage 3.

In case of a branch instruction a PFC (Program Flow Change) Signal is generated towards the previous section of the pipeline.

8 Operand Fetching

The fetching of operands is done by two stages. In Stage 3 the memory is addressed and in Stage 4 the data is received. While addressing, we have to check for Read-after-Write dependencies. Reading memory in Stage 3 might cause a conflict with Stage 6 where data is written back. For this we need some piece of hardware to control access to memory. We need one for each type of memory. They will be described in paragraph 8.3.

8.1 Operand addressing

The addressing is done in Stage 3. From the source information offered by stage 2, the address, memory type and the actual need of addressing can be extracted.

If addressing is done it is replied with an Advanced Ready Signal from the accessed memory. If this signal doesn't get active in the same clock cycle, Stage 1 must be stalled. The Advanced Ready Signal can get active late in the clock cycle. Not to let the Stall Signal be dependent of the Ready Signal the next incoming instruction can be buffered. Stage 1 then only needs to be stalled if the buffer is full.

To check if there is any dependency with an instruction present in Stage 4, Stage 5 or Stage 6 their destination addresses are checked with the source address of the instruction present in this stage. If there is a match with Stage 4 or Stage 5, this stage will wait and a stall is given to Stage 2 (next incoming instruction is buffered). If there is a match with Stage 6, data forwarding is done. The forwarded data is stored in the Source Information Word and the format is changed into 'no addressing'. After this it is, together with the instruction shifted towards Stage 4.

A possible stall from Stage 4 will get active early in the clock cycle. Therefore any addressing done in this stage can be stopped directly depending on this signal. If this stage stalls but Stage 4 doesn't, a NOP will be forced.

8.2 Operand receiving

Based on the information in the Source Information Word, Stage 4 receives the data from the addressed memory. Receiving data must be done when an addressing is performed. The data will only be available on the data bus for one cycle. If between the addressing in Stage 3 and the receipt of the data the Stall Signal from Stage 5 gets active, the data may not get lost. For this, stage 4 is able to buffer this data.

The source address is checked whether or not it is R0 or R1 from the active or an inactive bank. If it is, the data and some bank information are forwarded to Stage 2.

8.3 Accessing Memory

For each type of memory there are two stages who can try to access this memory. This may cause conflicts and therefore we need some kind of arbitration. This is done by a module for each type of memory, interfacing with the memory and the Stages.

In case of RAM and SFR space, Stage 3 and Stage 6 can try to access it at the same time. Priority is given to Stage 6 for the reason given in Chapter 3. In fact, the controllers for RAM and SFR are nothing more than multiplexers. The access of Stage 3 can now be delayed because of Stage 6. Therefore, if an access of Stage 3 succeeds it is announced with an Advanced Ready, also in case of an access of SFR space.

In case of ROM, Stage 3 and the ROM Module can both try to access it at the same time. Priority is given to Stage 3 and we only have to extend the ROM Module with a few multiplexers to handle this. In fact, the thing we are doing is letting Stage 3 place a request at the ROM Module, which will always be handled.

9 Instruction execution

When the operands have been fetched, the instructions are ready to be executed. To execute instructions, this stage must be able to perform the following operations:

- Arithmetic operations
- Logical operations
- Boolean variable manipulation
- Data transfer
- Conditional program branching

For the first three operations this stage needs an ALU. For the last operation it must be able to test certain conditions, for which it needs a test mechanism. Executing the instruction is done by switching multiplexers and other functional blocks of the stage according to the instruction code.

To be able to access all the internal SFR's, two busses, between Stage 5 and Stage 2 are available. One to transport source data to Stage 5 and one to transport the destination data back. The switching, to select the right data is done with the register information present in the source and destination information (retrieved in Stage 2).

9.1 The ALU

As already noted before, the ALU must be able to perform arithmetic operations, logical operations and Boolean variable manipulation. To save costs on data paths and multiplexers, it is also useful to let the ALU perform data transfer operations, which means nothing more than just switching the data through.

To simplify the design, the ALU is build up of separate blocks, all with a specific function. We could divide the functionality among those blocks according to the three different operations, but if we look at the multiply and divide instructions, a better division is possible:

In this core the multiply is executed in eight clock cycles. To track count a 3-bit counter register is used that is incremented every clock cycle. The ACCU is shifted every cycle, and depending of the bit, which is shifted out, Register B is added to a temporary register. In order to perform a multiply, three basic operations are necessary: addition, Shifting and incrementing.

According to this and the other necessary functionality of the ALU, it is split up in the following units:

- Shifting unit
- Logical unit

Incrementing and Decrementing unit
 Add and Subtract unit
 Bit Module
 Multiply and divide unit

The Bit Module is used to handle all bit operations. It works on byte level which means that it also extract the bit from the operand byte and puts it back again after operation. The Multiply and Divide unit makes use of the shift, increment and add functions of the other units. Figure 9.1 gives a simplified representation of this ALU.

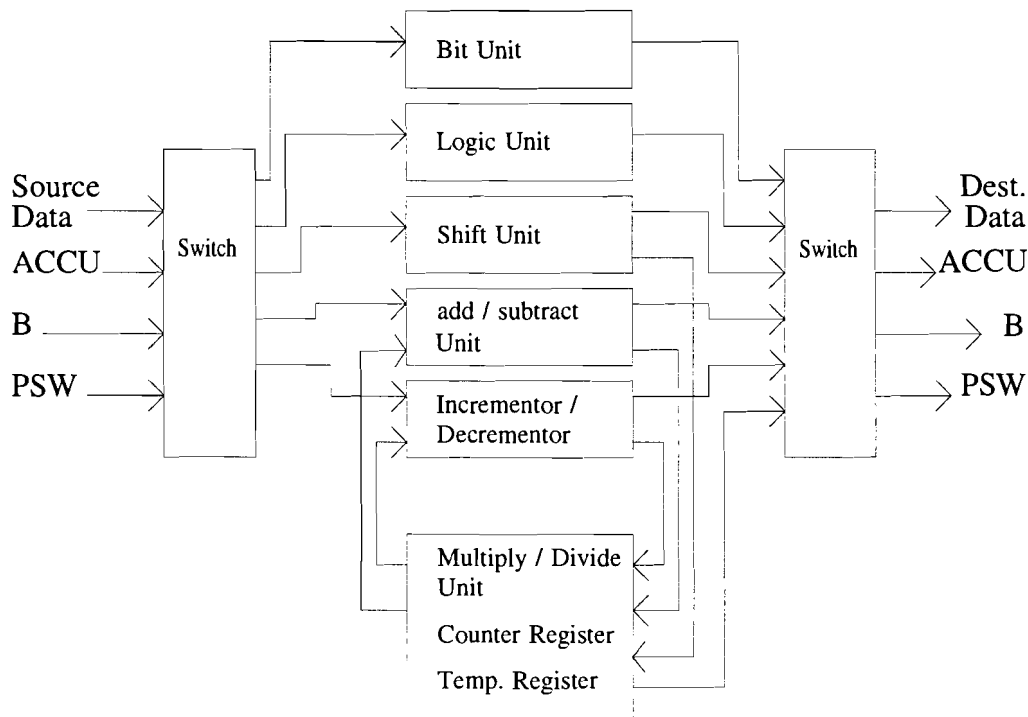


Fig. 9.1 ALU

9.2 Testing conditional branches

In order to decide whether a branch should be taken or not, a condition must be tested. Below the conditions and the necessary tests are given:

Bit value. The value of the bit is extracted from the operand byte by the Bit Module in the ALU. The branch is taken according to this value. Both bit and complement bit testing is possible.

Carry. The Carry is available in the PSW and depending on its value a branch is taken.

Accumulator Zero. Via some logic a Zero bit is retrieved from the ACCU and

branching is done according to its value.

Compare Jump Not Equal. For this condition two data objects must be compared. This is done by the logic unit in the ALU and depending on this information the branch is taken.

Decrement Jump Not Zero. For this condition a data object must be decremented by the Increment and Decrement unit in the ALU. This unit also has got the possibility to test the decremented value and according to this information the branch is taken.

In the figure below the test module is given.

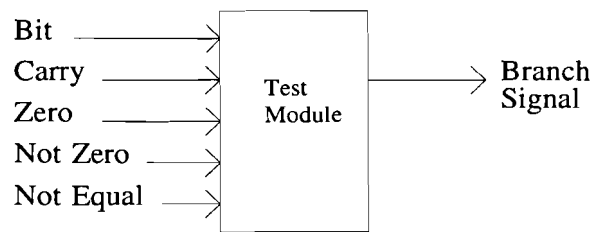


Fig. 9.2 test module

According to the outcome of the test, the PC is loaded with the target address of the branch instruction.

10 Write back stage

From Stage 5 the destination information and data are shifted towards Stage 6, the last stage of the pipeline. Here the data is written back to memory.

From the destination information the address and the type of memory to be accessed is extracted. The Read Signal is generated and if the access is successful, an Advanced Ready Signal is received. The latter is not necessary in case of writing to SFR space because a write to this memory is always accepted immediately.

In case of accessing RAM the Ready Signal might get active late in the clock cycle and therefore the incoming data and destination information is to be buffered. In this way the Stall Signal towards Stage 5 is not dependent of the Ready Signal.

The destination address is also checked whether it is pointing at R0 or R1 of any Register Bank. If so, the destination data, bank number and a register number are sent towards Stage 2. Figure 10.1 shows a simplified schematic of the stage.

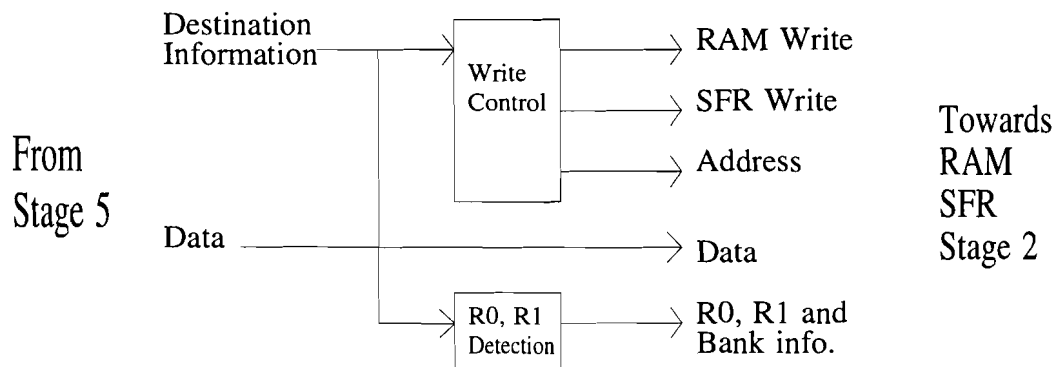


Fig. 10.1 Write Back

11 System Extension, Reduction and Performance

In this chapter a few ideas are given to extend or reduce the system. These ideas arose while designing it, but were not implemented. At the end of this chapter an estimation is made of the system performance. It is not meant as a guarantee but only to get an impression.

11.1 System Extension

The performance of pipeline as it is implemented now, can be improved by changing the Branch Handling of conditional branches and Data Forwarding. In both cases extra hardware is needed.

Branch Handling of conditional branches

If a branch takes place now, the addressing is stopped and the buffers in the ROM module are flushed. If the branch is taken, the PC is loaded with the target address and if the branch is not taken, the PC is, in fact, loaded with the Program Counter Value added to the branch instruction in Stage 2. This solution is simple to implement but the available hardware is not used optimally.

A better solution would be to let the fetching of instruction bytes continue and only flush the pipeline if a branch is taken. The problem with this is that instructions might influence the state of the pipeline and this is difficult to withdraw. But this can only happen in Stage 2 and further. So, it must be possible to perform some kind of instruction prefetching by the ROM Module and Stage 1. If a branch is taken, the present instruction bytes will be flushed and if not, the program flow can continue.

In order to do this, the state machine in Stage 1 needs to be changed. It must have the possibility to flush by clearing the Stage Registers behind it. This shouldn't be too difficult to implement. The ROM module already can perform flushing. In case of a branch, only Stage 1 needs to be stalled until a decision is taken. The ROM Module and Stage 1 will then be filled with instruction bytes.

Data Forwarding

Data is now only forwarded to Stage 3. In case of a hazard the maximum delay can still be 2 clock cycles.

In Stage 3 the source address is compared with the destination addresses of the preceding instructions. If there is a match with the instruction in Stage 4 or 5, Stage 3 is stalled. If there is a match with the instruction of Stage 6, data forwarding is done.

If we would also give Stage 4 and 5 the possibility to compare the source address of the

present instruction with the destination address of the instruction of Stage 6 the delay can be reduced to zero. If in Stage 3 a match is detected with Stage 4 or 5 the instruction will continue, but no addressing is done. If in Stage 4 a match is detected with Stage 6, data forwarding is done towards Stage 4, if not, the instruction will continue. If in stage 5 a match is detected data forwarding is done. If no match is detected, data forwarding already was performed and the data is available as usual.

PC incrementing

Incrementing the PC is once every 16 times done with the 16-bit ALU of Stage 2. If the PC Module has its own 16-bit incrementor, Stage 2 won't be interrupted anymore and the performance will increase. This extension will also simplify the design.

Interrupt mechanism

At this moment no interrupt mechanism is implemented. The core should be able to handle seven interrupts. The priority and masking will be taken care of outside the core. The only thing the core should be able to do is recognise a change in a three bit code and perform a CALL towards one of the seven interrupt addresses. The interrupt address is dependent of the interrupt number. A signal should be generated if an interrupt handling is started. If a RETI is performed, a signal should be given about the end of an interrupt.

The best way to implement this interrupt mechanism is to make a separate Interrupt Module. This module will provide a signal and an address towards Stage 1. In Stage 1 the Finite State Machine is extended in such way that after every instruction it checks whether an interrupt happened. If so, it forces a CALL with the right address and generates a signal, which informs about the start of the interrupt handling. If the receipt of an RETI instruction is detected, a signal is given to inform about the end of the interrupt routine.

11.2 System Reduction

The core will be used embedded in the CAN controller. The space available is limited. In order to reduce the die area of the core, some reduction can be done. Of course, the performance of the system will decrease.

The use of DPTR and SP as source in Stage 5

If the DPTR or SP is used as source via direct addressing, their data is retrieved by Stage 5 using a bus and some multiplexers. It is also possible to carry the data to Stage 5 via the Source Information Word.

The only problem is that in this case we have to check for a read-after-write hazard, which can be considered not to be totally fair. A Stall can be generated in stage 2 because of data, which is not going to be used until the instruction reaches Stage 5, after 3 clock cycles.

This solution saves the cost of the bus and some multiplexers, but introduces a much more

complex source generation. Also the Stalling Mechanism has got to be adjusted.

Stage buffering

In order to make the Stall mechanism independent from the advanced ready signals used by the memories, Stage 3 and 6 are buffered. These buffers can be taken away, but then also the clock frequency must be decreased. So, the performance of the system will decrease. This is heavily dependent of the timing of the advanced ready signals.

Simplifying the Shadow Register Update Mechanism

To update the Shadow Registers, at this moment a data forwarding towards Stage 2 is done from Stage 4 and Stage 6. The data forwarding from Stage 6 can be left out. It is then necessary to update the Shadow Registers every time a write towards the PSW, R0 or R1 is done. The Bank Number, which is stored together with the Ri value, can also be left out. It is sufficient to mark the Shadow register as valid or invalid.

11.3 System performance

The performance of a pipelined system is always dependent of the program code executed. If the programmer or the assembler is familiar with the design of the processor, a great increase in performance can be realised. In spite of this dependency, an estimation of the performance will be made in this paragraph.

If we want to get an impression of the increase of performance due to the pipelining technique, we have to make a comparison with the standard 8051. The performance will be given in MIPS (Million Instructions Per Second).

The standard runs at a frequency of 12 MHz and uses 12, 24 or 48 clock cycles for one instruction. The performance will be somewhere between 0.25 and 1 MIPS, but never better. Nowadays the standard is also available at 25 MHz, which increases the performance to maximally 2 MIPS.

The pipelined version is designed to run at a frequency of 33 MHz. Simulated in IDaSS it was able to perform all the instructions at a speed of 40 MHz, but for now we will assume 33Mhz. In the best case we can handle one instruction every clock cycle. This would give a performance of 33 MIPS. In the worst case we can have a program consisting of only conditional branches of three bytes each. In this case it would take eight clock cycles for an instruction to finish. The performance would then be $(33/8=)$ 4.125 MIPS. During testing the average time between the finishing of instructions was about 3 cycles. This means a performance of 10 MIPS. If the used program code would be programmed with the knowledge of the core's architecture, and dependencies would be avoided as much as possible, the performance might get as high as 15 MIPS (Running at 40 MHz even 20 MIPS).

As already said before, the performance can be increased when the programmer is familiar with the architecture of the core. In order to simplify programming a 'penalty table' can be

used. In appendix 6 a penalty table is given of the implemented 8051. In the table itself some delays are given. If an instruction is stalled, we get some kind of traffic light situation. If the cars have to wait, the space between the cars is reduced to a minimum. In the pipeline, if an instruction is stalled, the space between instructions will also disappear. This is why a lot of rules have to be added to the table in order to make it correct. Because of this the table will be very difficult to use. It can be used as guide but it surely isn't a guarantee. Exact calculation of the delay should be done using a simulation model.

12 Conclusions and recommendations

12.1 Conclusions

The microcontroller core is implemented in IDaSS as it is described in the thesis. Depending on the application it will be used for, the design can be reduced or extended as described in the previous chapter. Of course, it can also be used embedded in different applications than the CAN controller. For now it is a balance between performance and cost.

The clock frequency is about 40MHz so, the desired speed of 33MHz is easily reached. The performance is a factor 10 higher than the standard and the whole design is tested. All the instructions give the right result and also the dependencies are dealt with correctly.

The Interrupt module is not implemented yet. Implementing it like described in the previous chapter shouldn't give any problem. It only needs some adjustments in the State Machine.

12.2 Recommendations

The 8-bits ALU implemented in Stage 5 is taken from an existing design. The design doesn't use the hardware efficiently. By using a different design this can be improved.

The maximum clock speed is now limited by the time required by Stage 2. If the clock speed needs to be increased, this is the place to optimise. A possible cause for this bottleneck might be the 16-bit ALU.

To increase performance or reduce cost, the ideas described in the previous chapter can be used.

References

- [CRAG96] Cragon, H.G.
MEMORY SYSTEMS AND PIPELINED PROCESSORS.
Boston: Jones and Barlett Publishers, 1996.
- [EMMA87] Emma, P.G. and E.S. Davidson
CHARACTERIZATION OF BRANCH AND DATA DEPENDENCIES IN
PROGRAMS FOR EVALUATING PIPELINE PERFORMANCE.
IEEE Transactions on Computers, Vol. C-36 (1987), No. 7, p. 859-875.
- [FLYN95] Flynn, M.J.
COMPUTER ARCHITECTURE: PIPELINED AND PARALLEL
PROCESSOR DESIGN.
Boston: Jones and Barlett Publishers, 1995.
- [GONZ93] Gonzalez, A.M.
A SURVEY OF BRANCH TECHNIQUES IN PIPELINED PROCESSORS.
Microprocessing and Microprogramming, Vol. 36 (1993), No. 5, p. 243-257.
- [HOFM96] Hofman, M.
DESIGN OF A HARDWARE/FIRMWARE SOLUTION FOR CAN AND ITS
APPLICATION LAYERS.
Report: I11, Stan Ackermans Institute, Eindhoven, 1996.
- [HUAN93] Huang, Ing-Jer and A.M. Despain
EXTENDED CLASSIFICATION OF INTER-INSTRUCTION DEPENDENCY
AND ITS APPLICATION IN AUTOMATIC SYNTHESIS OF PIPELINED
PROCESSORS.
In: Proceedings of the 26th Annual International Symposium on
Microarchitecture. Austin, TX, USA, 1994.
Los Alamitos, CA, USA: IEEE Computer Society Press, 1994. P. 236-246.
- [INTE83] MICROCONTROLLER HANDBOOK.
Intel, 1993.
- [KOGG81] Kogge, P.M.
THE ARCHITECTURE OF PIPELINED COMPUTERS.
New York: McGraw-Hill, 1981.
- [LILJ88] Lilja, D.J.
REDUCING THE BRANCH PENALTY IN PIPELINED PROCESSORS.
Computer, Vol. 21 (1988), No. 7, p. 47-55.

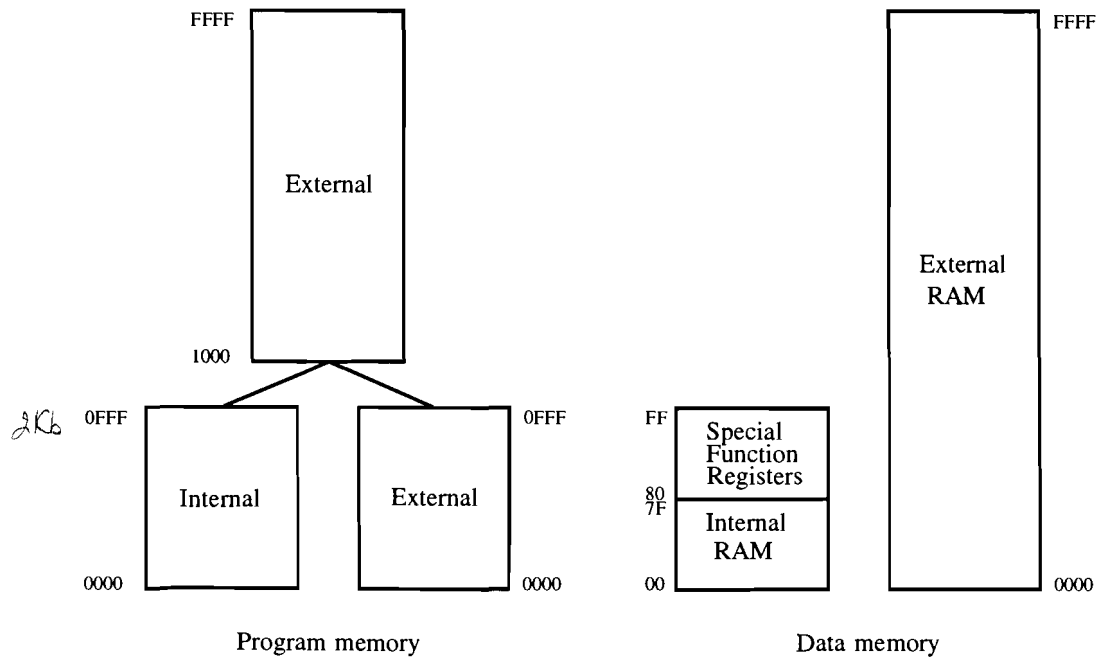
- [SLEG96] Slegers, S.
THE DESIGN AND IMPLEMENTATION OF AN EXTERNAL INTERFACE
FOR A CAN/CAL CONTROLLER.
Master's Thesis: EB 637, Technical University of Eindhoven, Faculty of
Electrical Engineering, 1996.
- [VERS90] Verschueren, A.C.
IDASS FOR ULSI - INTERACTIVE DESIGN AND SIMULATION SYSTEM
FOR ULTRA LARGE SCALE INTEGRATION.
Eindhoven University of Technology, Section of Information and
Communication Systems, 1990.
- [VERS94] Verschueren, A.C.
TOWARDS A HIGHLY PIPELINED 8051 - PRELIMINARY
INVESTIGATIONS.....
Eindhoven University of Technology, Section of Information and
Communication Systems, 1994.

Appendix 1 Standard instruction set

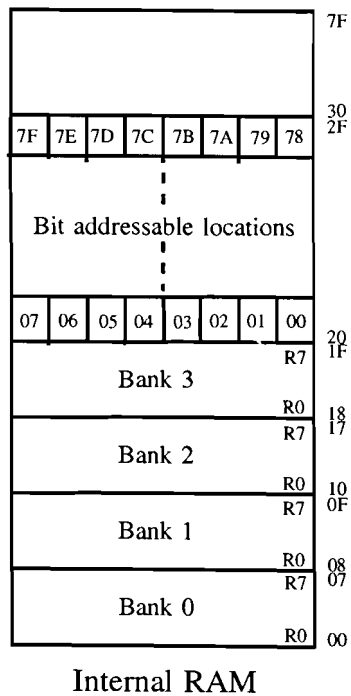
In this table white indicates 1-byte instructions, light grey indicates 2-byte instructions and darker grey indicate 3-byte instructions.

	x0	x1	x2	x3	x4	x5	x6..x7	x8..xF
0x	NOP	AJMP page 0	LJMP a16	RR A	INC A	INC dir	INC @Ri	INC Rn
1x	JBC bit,rel	ACALL page 0	LCALL a16	RRC A	DEC A	DEC dir	DEC @Ri	DEC Rn
2x	JB bit,rel	AJMP page 1	RET	RL A	ADD A,#d8	ADD A,dir	ADD A,@Ri	ADD A,Rn
3x	JNB bit,rel	ACALL page 1	RETI	RLC A	ADDC A,#d8	ADDC A,dir	ADDC A,@Ri	ADDC A,Rn
4x	JC rel	AJMP page 2	ORL dir,A	ORL dir,#d8	ORL A,#d8	ORL A,dir	ORL A,@Ri	ORL A,Rn
5x	JNC rel	ACALL page 2	ANL dir,A	ANL dir,#d8	ANL A,#d8	ANL A,dir	ANL A,@Ri	ANL A,Rn
6x	JZ rel	AJMP page 3	XRL dir,A	XRL dir,#d8	XRL A,#d8	XRL A,dir	XRL A,@Ri	XRL A,Rn
7x	JNZ rel	ACALL page 3	ORL C,bit	JMP @A+DPTR	MOV A,#d8	MOV dir,#d8	MOV @Ri,#d8	MOV Rn,#d8
8x	SJMP rel	AJMP page 4	ANL C,bit	MOVC A,@A+PC	DIV AB	MOV dir,dir	MOV dir,@Ri	MOV dir,Rn
9x	MOV DPTR,#d16	ACALL page 4	MOV bit,C	MOVC A,@A+DPTR	SUBB A,#d8	SUBB A,dir	SUBB A,@Ri	SUBB A,Rn
Ax	ORL C,/bit	AJMP page 5	MOV C,bit	INC DPTR	MUL AB	Reserved	MOV @Ri,dir	MOV Rn,dir
Bx	ANL C,/bit	ACALL page 5	CPL bit	CPL C	CJNE A,#d8,rel	CJNE A,dir,rel	CJNE @Ri,#d8,rel	CJNE Rn,#d8,rel
Cx	PUSH dir	AJMP page 6	CLR bit	CLR C	SWAP A	XCH A,dir	XCH A,@Ri	XCH A,Rn
Dx	POP dir	ACALL page 6	SETB bit	SETB C	DA A	DJNZ dir,rel	XCHD A,@Ri	DJNZ Rn,rel
Ex	MOVX A,@DPTR	AJMP page 7	MOVX A,@R0	MOVX A,@R1	CLR A	MOV A,dir	MOV A,@Ri	MOV A,Rn
Fx	MOVX @DPTR,A	ACALL page 7	MOVX @R0,A	MOVX @R1,A	CPL A	MOV dir,A	MOV @Ri,A	MOV Rn,A

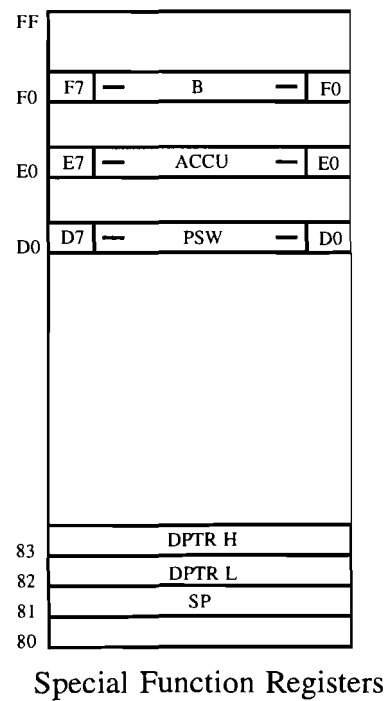
Appendix 2 Standard memory map



8051 memory map

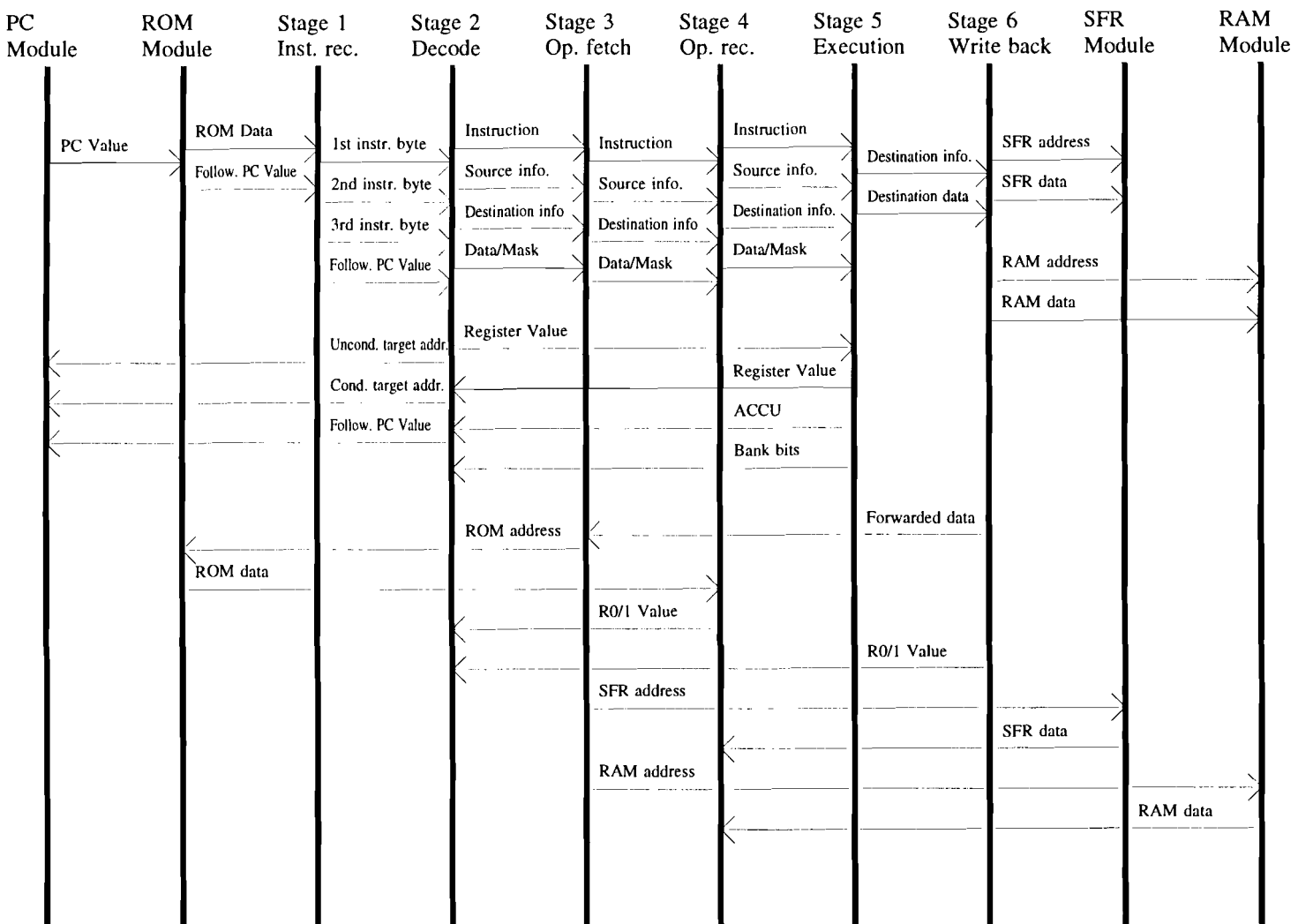


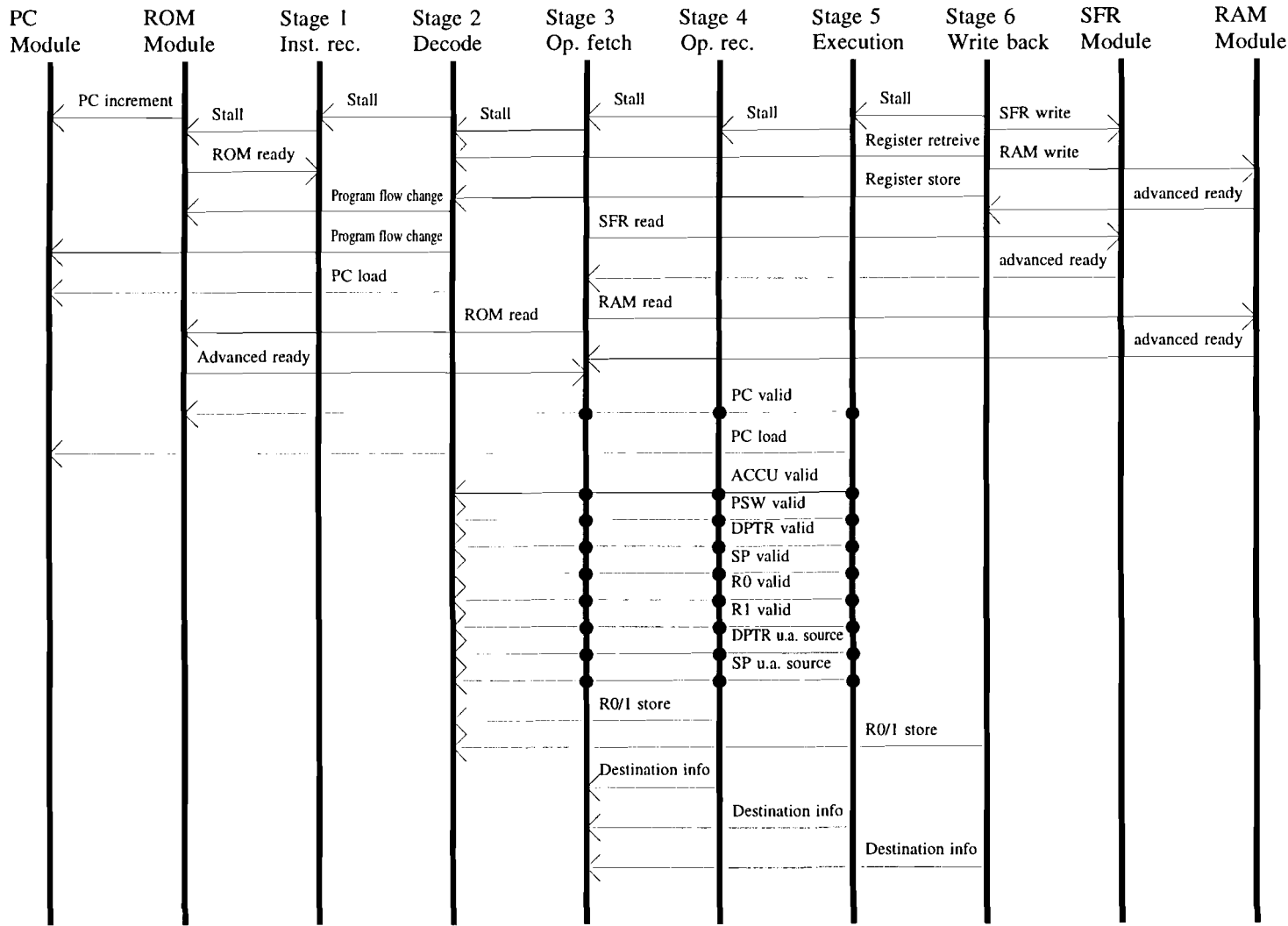
Internal RAM



Special Function Registers

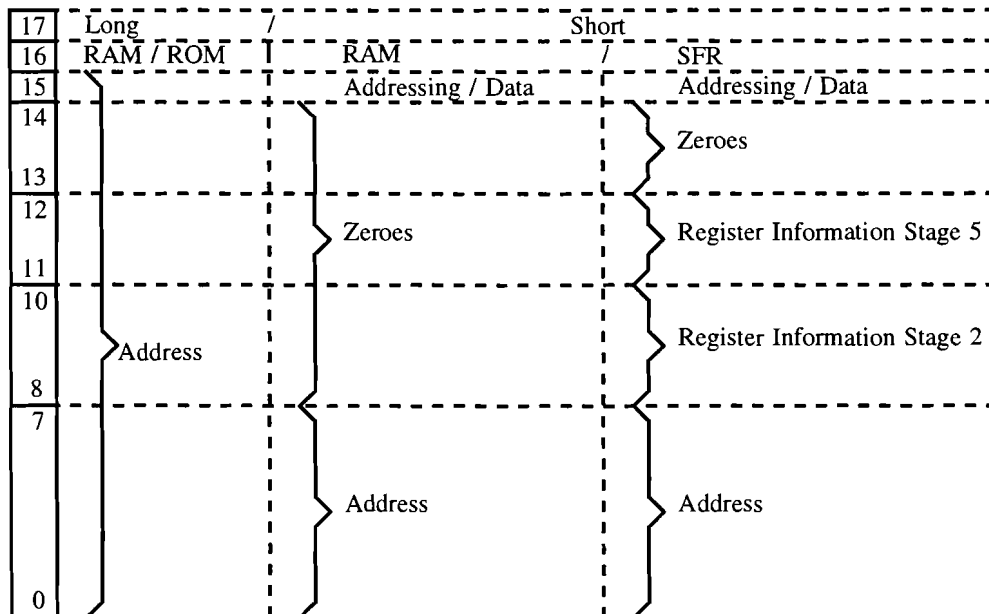
Appendix 3 Data and address flow in the pipeline



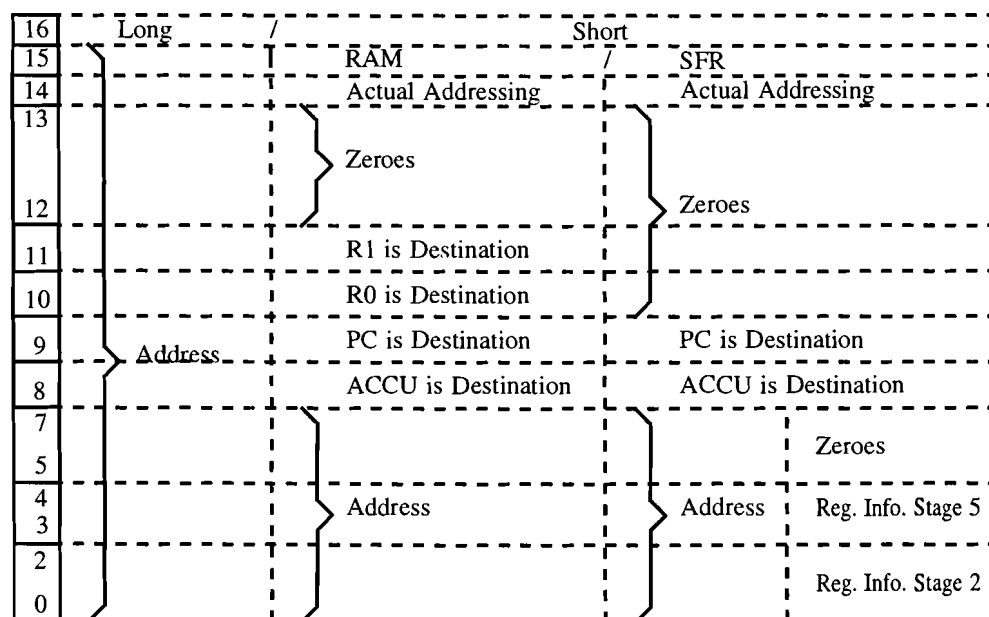


Appendix 5 Source and destination information format

Source Information Format



Destination Information Format



In these formats, register information is included. This information is necessary to detect dependencies and access the registers in case they are source or destination operands. It has

got the following format:

Register Information concerning registers kept in Stage 2:


000	no register selected
001	no register selected
010	Data Pointer Low Byte
011	Data Pointer High Byte
100	Program Counter Low Byte
101	Program Counter High Byte
110	Stack Pointer Low Byte
111	Stack Pointer High Byte

Register Information concerning registers kept in Stage 5:

00	no register selected
01	B Register
10	Accumulator
11	Program Status Word

Appendix 6 Penalty Table

	Previous Instruction									
	Condition branch	Unconditional branch	Mem. dest. (No int. Reg)	ACCU as destination	PSW as destination	DPTR as dest. via dir.	SP as dest. via dir.	DPTR as src. via dir.	SP as src. via dir.	Ri as destination
ACCU used in addr. calc.	2	5	0	3	0	0	0	0	0	0
DPTR used in addr. calc.	2	5	0	0	0	3	0	0	0	0
DPTR as dest. in Stage 2	2	5	0	0	0	3	0	3	0	0
Rn addr. as destination	2	5	0	0	3	0	0	0	0	0
Rn addr. as source	2	5	2*	0	3	0	0	0	0	0
Dir. addr. as source	2	5	2*	0	0	0	0	0	0	0
@Ri addr. as destination	2	5	0	0	3 (+2)	0	0	0	0	4
@Ri addr. as source	2	5	2*	0	3 (+2)	0	0	0	0	4
SP as src. and dest. in Stage 2	2	5	0	0	0	0	3	0	3	0
All other instructions	2	5	0	0	0	0	0	0	0	0


 This delay takes place in stage 3
 All other delays take place in stage 2

* This delay only happens if the source and destination address are the same
 (+2) Two more delay cycles if the Shadow Registers must be updated

The delays in the table assume continuous instruction fetch.

Rules and exceptions:

- Longest delay counts. Minimal delay is zero.
- A delay which is not caused by a branch, must be decremented with the number of instruction bytes between previous and actual instruction (program code compensation) plus the instruction bytes minus one of the actual instruction (instruction code compensation).
- If a previous instruction suffers a delay in stage 2, the instruction code compensation is decreased with the delay if the instruction is immediately succeeding. If the instruction is not immediately succeeding, the program code compensation is decreased with the number of bytes of its first instruction minus one with a maximum of the delay.
- If the previous instruction suffers a delay in stage 3, the instruction code compensation is decreased with the delay if the instruction is immediately succeeding. If the instruction is secondly succeeding, the program code compensation will be: Number of program code bytes decreased with the delay to a minimum of one. The instruction code compensation is decreased with the delay minus the decrease of the program code compensation. In case of the third succeeding instructions, the program code compensation will be: Number of program code bytes decreased with the delay with a minimum of the number of instructions it consists of.
- If the previous instruction suffers a delay (always seven) in stage 5 (mul and div) something similar as above can be used till the sixth succeeding instruction.
- Instructions who use RAM or SFR source addressing can suffer a delay of one in stage 3 due to a conflict with stage 6.
- Instructions which use the ALU of stage 2 can get a delay of one due to the incrementing of the program counter via this ALU.
- A CALL can be seen as a sequence of one JUMP and two pushes of the Program Counter Value. The pushes don't suffer a delay from the jump. The two pushes give an extra delay of two cycles.
- A RET is replaced by two pops towards the program counter. The pops give an extra delay of two cycles. The last pop gives a delay similar to a conditional jump.
- Reading from ROM in stage 3 causes a delay of one cycle in instruction byte fetching. This delay happens normally 3 cycles after the fetch of the reading instruction.