ICS/EB 685

Technische Universiteit Eindhoven

Faculty of Electrical Engineering
Section of Information and Communication Systems

Practical Training Report

# Implementation of the Interrupt Mechanism in a pipelined 8051 microcontroller core

E.A. Patent

| | |
|---|---|
| Coach: | dr.ir. A.C. Verschueren |
| Supervisor: | Prof.ir. M.P.J. Stevens |
| Period: | 20 January – 20 March 1998 |

The Faculty of Electrical Engineering of the Eindhoven University of Technology does not accept any responsibility regarding the contents of practical training reports

# Abstract

In this report there is a brief overview of the pipelining of Intel 8051 microcontroller core. The reason is the CAN (Controller Area Network) controller, which is being developed at the Section of Information and Communication Systems. In this CAN controller application processor will be a fast 8051 based microcontroller core, used pipeline technique.

One of the important characteristics of the developing core is interrupt handling ability. The core being developed must handle seven interrupts. Therefore interrupt theory first is considered. Interrupt Handling Mechanism is implemented in Stage 1, where instruction bytes are fetched from the ROM Module and offered to Stage 2. This is done in such a way, that after every instruction it checks whether an interrupt happened. If so, if forces a CALL with the address depending on the interrupt source and generates a signal, which informs about the start of interrupt handling. If RETI instruction is encountered, it generates a signal about the end of interrupt handling.

The implementation of the Interrupt Mechanism and testing are done in Interactive Design and Simulation Systems (IDaSS) for Ultra Large Scale Integration.

# Contents

# 1 Introduction

CAN (Controller Area Network) is a network used in the automotive industries [1]. Nowadays various companies include CAN controller 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 too much.

At the Eindhoven University of Technology, Faculty of Electrical Engineering, the Information and Communication System 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. In [1] it is assumed that integrated microcontroller should be an Intel 8051 core. To use this core in the CAN controller, some adjustments to the standard 8051 have to done. In [1], in order to increase the speed performance of the core, Intel 8051 compatible microcontroller is designed using a pipelining technique.

In this report a brief overview of pipelining technique is first considered. The problems that appear using this technique are described. One of the most essential characteristics of the core being developed is interrupt handling ability. In the core designed in [1] interrupt mechanism was not implemented.

This report describes the functional design and implementation of Interrupt Mechanism. Therefore interrupt theory [2] is discussed. The core should be able to handle seven interrupts. One of the possibility to implement the Interrupt Mechanism is to make certain adjustments in Stage 1, where the instruction is fetched from the ROM Module, and Stage 2, where instruction is decoded and addresses are calculated. In this case Interrupt Handling Mechanism after every instruction checks whether an interrupt occurs. If so, it forces a CALL with the address depending on the interrupt source and generates a signal which informs about the start of interrupt handling. If RETI instruction is encountered, it generates a signal about the end of interrupt handling.

The implementation of Interrupt Mechanism and testing are done in Interactive Design and Simulation System (IDaSS) for Ultra Large Scale Integration [3].

# 2 Pipelining the 8051 microcontroller core

Pipelining is a technique to improve system performance by pursuing instruction-level parallelism [1]. 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.
In the present work it is chosen that microcontroller will be Intel 8051 based.

## 2.1 Pipelining in general

The idea of pipelining is to split up the functions in more subfunctions [1]. The hardware used to evaluate such a subfunction is called a *stage*. This hardware normally does not contain any memory. Between the stages registers are included to keep a discrete and synchronized data flow.

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 subfunctions independent from the activities of the other stages. Now more instructions can be handled in the same time. However the pipelining almost never gives a performance, shown in theory.

There are some problems that influence the program flow through the processor. These problems can be divided in two categories [1]:

1. **Branching problems** or Program Flow Changes occur when **Jump** instruction, a **Call** or a **Return** is executed. In case of a conditional **Jump** the evaluation is done at the end of pipeline. The preceding part of the pipeline should be filled with an instruction flow depending on the outcome of the evaluation.

2. **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. Data dependent hazards happen if two instructions in different stages attempt to use the same data.

Another problem about pipelining is the additional hardware needed, and with this the additional cost. This makes the design of pipeline more like finding compromise between performance and costs.

## 2.2 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. Taking into account that addressing of the memory is done in two steps – addressing and receiving data, following partitioning is possible suitable for all instructions [1]:

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

It must be noted 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. According to this partitioning of the function of microcontroller, the pipeline is divided into following stages [1]:

| | |
|---|---|
| **Stage 0,**<br>**ROM Module.** | This stage addresses ROM. In the remainder of this theses it will be referred to as ROM Module |
| **Stage 1,**<br>**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,**<br>**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 address is valid. In order to do these calculations a 16-bit ALU is present. In this stage Data Pointer and Stack Pointer are kept |
| **Stage 3,**<br>**Operand Addressing.** | The addressing and 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,**<br>**Operand Receive.** | The operand is received and a kind of data forwarding is done to update the R0 and R1 Shadow Registers |
| **Stage 5,**<br>**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. |

Besides these stages for each memory type a module is added. Program Counter is represented as a separate module because of the handling of Program Flow Changes. In spite of microcontroller is Intel 8051 based, some adjustments to the memory map are made [1]. This is caused by application in CAN controller. For that reason the environment of the controller is provided by interface of the CAN controller. In the implementation microcontroller core has got the following interface with the environment:

**ROM:** Address (16 bit), Data in (8 bits), Read (1 bit), Ready (1 bit).

**RAM:** Address (16 bit), 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).

## 2.3 Dependencies

As mentioned above, one of the significant problems in a pipelined design is handling of dependencies. In a pipelined design instruction executions are overlapped and it is possible that operations involved in executing an instruction are not finished when operations involved in execution 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 [1]:

1. **Data dependent hazards** occur in cases if a data object is accessed by different instructions of which the execution overlaps. They are:

   -*Read-after-Write* (RAW) hazard. These hazards exist if an instruction wants to read but the previous instruction still has to write to the same data object.

   -*Write-after-Read* (WAR) hazard. These hazards exist if an instruction wants to write before the previous instruction has read to the same data object.

   -*Write-after-Write* (WAW) hazard. These hazards will exist if two instructions write to the same data object in the wrong order.

2. **A structural dependent hazards** occurs if two instructions try to use the same piece of hardware. In this application four pieces of hardware are shared: three of them are the memory access points (actually the only shared part is the address bus), and the fourth part - 16-bit ALU of stage 2. This happens, as shown further, because of incrementing of the Program Counter once every 16 times. Here there is a problem of priority, which can not be simply given to one of both sides.

## 2.4 Stalling of the pipeline

In every stage it can happen that an instruction stays for more than one clock cycle. A data dependent hazard, structural hazard or any other reason an instruction should need the facilities of a certain stage more than one cycle can cause this. If it happens [1], 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 and this will prevent structural hazards.

The technique to prevent structural hazards in case of any collision is called *stalling*. The signal used to inform a previous stage about this situation is called *stall.*

In this implementation decentralized stalling system is chosen [1]. It means that the responsibility of local stalling is given to each stage. If a stage is stalled, it gives a stall to the stage behind it. The hole system of stalling is in fact a chain of local stalls. This type of stall mechanism is called a *ripple stall.* A problem with a ripple stall is the time that the stall signal needs to pass through the system. A possible solution, offered in [1], is buffering the pipeline at critical points.

## 2.5 Branching

**CALL** and **RETURN** are control-flow-instructions, refer to brunch. They are used to change the sequence of the program. In order to execute a branch, two operations must be performed [1]:

- Computation of the target address and
- Evaluation of the condition that determines whether a branch should be taken (if it is a conditional branch).

According to [2], there are three classes of control transfer operations:

1. Unconditional **Calls**, **Returns**, and **Jumps**;
2. Conditional **Jumps**;
3. Interrupts.

All control transfer operations cause, some upon specific conditions, the program execution to continue at a non-sequential location in program memory [2].

**Unconditional Calls, Returns and Jumps** transfer control from the current Program Counter Value to the *target address.*
Absolute Call (**ACALL**) and Long Call (**LCALL**) push the address of the next instruction onto the stack and then transfer control to the target address. ACALL is a 2-byte instruction used in case if the target address is in the current 2K page. LCALL is a 3-byte instruction that address the full 64K program space.

**ACALL** increments the PC twice to obtain the address of the following instruction, then pushes the 16-bit result onto the stack (low-order byte first) and increments the stack pointer twice. The destination address is obtained successively concatenating the 5 high-order bits of the incremented PC, op-code bits 7-5, and the second byte of the instruction.

**LCALL** adds three to the Program Counter to generate the address of the next instruction and then pushes the 16-bit result onto the stack (low-order byte first), incrementing the stack pointer by two. The high-order and the low-order bytes of the PC are then loaded with the second and the third bytes of the LCALL instruction, respectively.

Return from subroutine (**RET**) transfers control to the return  address saved on the stack by a previous **Call** operation and decrements the stack pointer register by two. **RET** pops the high- and the low-order bytes of the PC successively from the stack, decrementing the stack pointer by two. Program execution continues at the resulting address, generally the instruction immediately following an ACALL or LCALL.

Absolute Jump (**AJMP**), Long Jump (**LJMP**), Short Jump (**SJMP**) transfer control to the target operand.
**AJMP** transfers program execution to the indicated address, which is formed at run-time by concatenating the high-order five bites of the PC (after incrementing the PC twice), op-code bits 7-5, and the second byte of the instruction. The destination must

be within the same 2K block of program memory as the first byte of the instruction following AJMP.

**LJMP** causes an unconditional branch to the indicated address, by loading the high- and low-order bytes of the PC with the second and third instruction bytes, respectively. The destination can be in the full 64K program memory address space.

**SJMP** instruction provides for transfers within a 256-byte range centered about the starting address of the next instruction (-128 to +127).

Jump indirect (**JMP @A+DPTR**) performs a jump relative to the DPTR register. The operand in A is used to the address in the DPTR register. Thus the effective destination for a jump can be anywhere in the program memory space.

## Conditional Jumps.

Conditional jumps perform a jump contingent upon a specific condition. The destination will be within a 256-byte range centered about the starting address of the next instruction (-128 to +127).

## Interrupt Returns.

Return from Interrupt (**RETI**) transfers control as does RET, but additionally enables interrupts of the current priority level.
**RETI** pops the high- and low-order of the PC successively from the stack, and restores the interrupt logic to accept additional interrupts at the same priority level as the one just processed. The stack pointer is left decremented by two. Program execution continues at the resulting address, which is generally the instruction immediately after the point at which the interrupt request was detected.

When executing a branch some hazards may occur [1], which due to the frequent use of branches might cause a substantial reduction of processor performance. There are several ways to reduce the cost of branches. In this implementation [1] *Early Computation of the branch* is used. This means that the computation of the target address must be placed in the beginning of the pipeline. In Stage 2 for that reason a 16-bit ALU is needed to calculate the operand addresses. 16-bit ALU will also be used to calculate the branch target address. Before an instruction is recognized as a branch, it must be fetched from memory and partly decoded (see also further). A branch can cause a delay of 3,4 or 5 clock cycles in case of an unconditional branch and 7 or 8 in case of a conditional branch [1].

# 3 Interrupt Theory

One of the most essential characteristics of the developing core is interrupt handle ability. In this implementation the core should be able to handle seven interrupts. The priority and masking must be taken care outside of the core. The only thing the core should be able to do – is to recognize a change in a three bit code and perform a CALL towards one of the seven interrupt addresses. The interrupt address is dependent on the interrupt number. Also a signal should be generated if an interrupt handling is started. If a RETI is encountered, a signal should be given about the end of an interrupt. In this chapter first an overview of interrupt theory is given. After this interrupt handling in the implementation is considered.

## 3.1 Priority level structure

Each of interrupt sources can be programmed to one or two priority levels [2]. The interrupt sources are from the two external interrupt inputs; one each from the three timer/counter overflow flags and one from the serial port.

Each source can be individually enabled or disabled by setting or clearing a bit in Special Function Register IE, and also can be programmed to a high-priority level or low-priority level by setting or clearing a bit in Special Function Register IP. All of interrupt flags can be set or cleared by software with the same effect as if by hardware.

A low-priority interrupt can itself be interrupted by a high-priority interrupt, but not by another low-priority interrupt. A high-priority interrupt cannot be interrupted. To implement these rules, the interrupt system contains two non-addressable "priority level active" "flip-flops". One indicates that a high-priority interrupt is being serviced, and blocks all further interrupts. The other indicates that a low-priority interrupt is being serviced, and blocks all but high-priority interrupts.

In the event that requests of the same priority level are received simultaneously, an internal polling sequence determines which request is serviced. Thus, within each priority level there is a second priority structure.

All the interrupt sources are examined sequentially during each cycle, such that by S6 of any cycle (at the sixth state of the machine cycle [2]) all active interrupt requests have been found and prioritized. Response to the active request of highest priority will commence with state '1' of the next machine cycle, provided the response is not blocked by any of the following conditions:

1. An interrupt of equal or higher priority level is already in progress.

2. The current machine cycle is not the final cycle in the execution of the instruction in progress. (In other words, *no interrupt request will be responded to until the instruction in progress is completed*).

3. The instruction in progress is RETI or an access to Special Function Register IE or IP. (In other words *no interrupt request will be responded to after RETI or after read or write to IE or IP until at least one other instruction has been executed*).

If any of the above conditions exists, the result of the interrupt poll is discarded. If none of the above conditions exists, the result of the interrupt poll is acted on with the very next machine cycle.

## 3.2 Interrupt Response Protocol

The processor acknowledges a request by first setting the appropriate "priority level active" flip-flop. Then it executes a hardware subroutine call to servicing routine. It also clears the flag that requested the interrupt. The hardware subroutine call pushes the contents of the Program Counter onto the stack (but it does not save the PSW) and reloads the PC with an address that depends on the source of the interrupt request, as shown below:

| **Interrupt number** | **Vector location** |
|---|---|
| 1 | 0003 H |
| 2 | 000B H |
| 3 | 0013 H |
| 4 | 001B H |
| 5 | 0023 H |
| 6 | 002B H |
| 7 | 0033 H |

(Standard Intel 8051 microcontroller according to [2] has only five interrupt sources). Execution proceeds from that address until the RETI instruction is encountered. The RETI instruction clears the "priority level active" flip-flop that was set when this interrupt was acknowledged. Then it pops the bytes from the stack and reloads the Program Counter. Execution of the interrupt program commences from where it left off.

## 3.3 Interrupt Handling in the Application

As it was noted before, the priority and masking are taken care outside of the core. For that reason a special block can be used – Interrupt Controller. From this block the core receives an Interrupt Request Signal, which is 3-bit long and generates a signal – Interrupt Acknowledge, which is 2-bit long towards Interrupt Controller if an interrupt handling is started.

One of the possibilities to implement this is to make changes in Stage 1. In this stage bytes coming from the ROM Module are put together in right registers and complete instruction is offered to Stage 2. After every instruction it can be checked whether an interrupt happened. If so, a CALL should be forced with the right address and generates a signal, which inform about the start of the interrupt handling. As soon as RETI instruction is encountered, a signal should be given to inform about the end of interrupt routine.

# 4 Stage 1 – Instruction Fetch

In order to run a program, instruction bytes have to be fetched from program memory (ROM) [1]. 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. If interrupt is happened a signal must be given about the start of the interrupt handling. The ROM Module, Program Counter Module and Stage 1 do all this. In this chapter a brief overview of instruction fetching is given. After that main attention is paid on the Interrupt Handling Mechanism.

## 4.1 ROM Module

The ROM Module [1] does the actual addressing of ROM. To address the ROM an address (which is the Program Counter) and a ROM Read Signal are used.

1. The address is selected via special multiplexer, controlled by the Stage 3 Request Signal (which get active if the addressing must be done by Stage 3). Selection is made from two addresses: Program Counter and address given by Stage 3.

2. The Read Signal is made up from several signals. Those signals are coming directly from registers. This means that they are stable at the beginning of the cycle. (Exceptions are PFC- signal and Request from Stage 3). From Stage 1 the Read Signal is controlled by the Stall Signal. In order to solve problems caused by arising of Stall Signal (which is actually a ripple Stall and has a property to get active late in the clock cycle), Read Signal is made to be not directly dependent of the Stall Signal. This can be done by buffering the input from ROM [1]. In this implementation it is chosen to finish already started access.

In case of branches and Program Counter dependent addressing, the increased PC Value is needed to calculate target or operand address. For that reason the ROM Module sends with each instruction byte the Increased Program Counter Value (pointing at the next instruction) toward Stage 1 (Fig. 1).

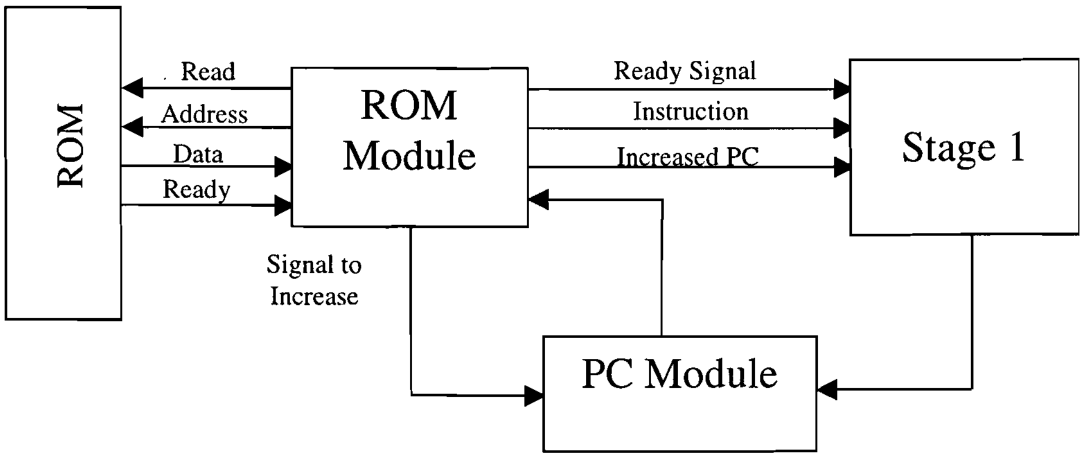


Fig. 1. Relations between Stage 1, ROM Module and PC Module

The data coming from ROM is said to be valid the clock cycle after an Advanced Ready Signal is given. This signal is used to inform Stage 3 in case the addressing is done in request of Stage 3. Else the Advanced Ready Signal is used to force the Program Counter to increase. Towards Stage 1 ROM Module offers the instruction byte with accompanying Ready Signal.

## 4.2 Program Counter Module

The Program Counter Module is controlling the access to, and the holding of Program Counter. The different possible actions to be taken by the logic are: holding the Program Counter, storing a value from Stage 5 or Stage 2 and making sure the Program Counter is loaded with the right value after PFC (Program Flow Change) is activated.

Every time an addressing is done Program Counter is increased. Every time a branch instruction is executed, it must be updated with the right value [1].

Incrementing is done with an incrementor. To keep the cost of hardware low, Program Counter Module uses a 4-bit incrementor. Every 16 increments the Program Counter is once incremented by the 16-bit ALU of Stage 2.

In order to decide whether a branch should be taken or not, a condition is tested in Stage 5. According to the outcome of the test, the Program Counter is loaded with the target address of the branch instruction.

In Stage 2 in order to speed up performance, instruction resolving can be done for number of instructions. Among of them – instructions, concerning unconditional branches - **AJMP, LJMP, SJMP, JMP @A+DPTR**. In this case the target address is calculated and the Program Counter is updated. In case of a branch instruction a PFC signal is generated towards previous stages of the pipeline.

Signals from Stage 5 and Stage 2 can never happen at the same time because they load the Program Counter in case of a Program Flow Change. A PFC is always then active. It must be noticed that in case of branches, the Program Counter loaded with PC Value kept with branch instruction. If branch is taken, Program Counter is loaded with the target address under control of Stage 2 (unconditional branch) or Stage 5 (conditional branch).

## 4.3 Stage 1. Interrupt Handling Mechanism.

After addressing is done, Stage 1 receives instruction bytes from the ROM Module [1]. Such an instruction byte can be the first, the second or the third byte of an instruction. The function of Stage 1 is to collect different bytes coming from the ROM Module in a parallel format for easy handling in the instruction pipeline. *The first instruction byte* is placed in register one and two. This is done in order to handle **Absolute Jumps** and **Long Jumps** in a similar way in Stage 2. *The second byte* is placed in register two or three, depending on the instruction involved. *The third byte* is placed in register three or two respectively.

In order to arrange instruction bytes, Stage 1 uses a Finite State Machine [1]. According to the first byte received, Finite State Machine places another bytes (if they are available) in the corresponding registers. This State Machine has four states (of which the fourth state can also be considered as two states).

Let's discuss the operation of the Stage 1 with regard to interrupt handling.

The instruction set of pipelined 8051 microcontroller includes single-byte, two-byte and three-byte instructions. The instruction op code format [2] consists of a function mnemonic followed by a "destination, source" operand field. This field specifies the data type and addressing method(s) to be used.

In this implementation in order to save hardware costs and to simplify the design of the next stages, some instructions are replaced [1] with the sequence of other instructions. Among of them – **Call**, replaced by a **Jump** and two **Pushes** of the Program Counter (high and low byte), and **Return**, replaced by two **Pops** toward the Program Counter.

Before receiving the first byte of the following instruction the control connector of Register 1 has a value, which activate a Load function of the register. If there is no ROM Ready Signal, NOP is loaded. Initially State Machine is in the state number 1.

When the first byte of the instruction is received, it is decoded. Decoding of each byte of instruction is done with the use of constant generators Dec 0, Dec 1, Dec 2, which set 'zero' or 'one' on their outputs according to the instruction byte on their inputs. The operator 'Concat' concatenates these values and this three-bit code control the state of Finite State Machine.

The Interrupt Request Checking Test can be performed in 3 states (after single-byte, two-byte and three-byte instructions are completed). In those states instruction is offered to the next stage. Before fetching the first byte of the next instruction, the Finite State Machine can check whether an interrupt occurs. If so, the Program Counter Value, pointing on the next instruction, is pushed onto the stack and PC is loaded with the target address, depending on the interrupt source. Interrupt Acknowledge Signal is then generated and the core executes an interrupt routine. To implement this, functions 'one', 'six' and 'ten' of the original Finite State Machine should be changed in such a way, to perform an Interrupt Checking Test.

However this method gives good results (pushing the **correct** value of the Program Counter onto the stack and, respectively, return to the right address after RETI is encountered) only for instructions, which do not perform operations with the Program Counter. For example – NOP. Problems appear in case of interrupting, for instance, a **Jump** instruction. A wrong value of the Program Counter is pushed onto the stack and program execution continues at the wrong address.

One of the ways to avoid these problems is to check whether an interrupt occurs only in the state number 1, where the first byte of the following instruction is received. It means that if an Interrupt Request Signal arises, Stage 1 must ignore received first byte of the instruction, decrement the Program Counter Value making it pointing on the same byte, and push *this* value onto the stack.

To implement this we have to make certain changes in Stage 1:

1. In the original Finite State Machine **functions 'one' – 'five'** must be upgraded in such a way, to perform an Interrupt Request Checking Test and a CALL towards one of the seven interrupt addresses. In the same time it is not necessary to do that in the state number 2 and in the state number 3, as in previous version. Description of all functions can be found in Tables 2 – 14 in Appendix 1.

2. **Function 'zero'** must be created and state 0 must be added to the original Finite State Machine. In the implementation [1] it is assumed to perform NOP when current instruction is already fetched from the ROM Module and no ROM Ready Signal received (it means to write value $00 into the register 1) (functions 'one', 'six', and 'ten'). This is made in order to inform next stages of the pipeline to wait until ROM Ready Signal appears. The main idea to insert additional state is to make difference between *this* NOP and a NOP loaded from the program. In this case if the instruction is completed and there is no ROM Ready Signal, Finite State Machine is switched to state 0 (**function 'zero'**). As soon as ROM Ready Signal arises, first byte of an instruction is loaded to the register 1 and decoded. Description of this function can also be found in Table 1 in Appendix 1.

3. Decrementing of Program Counter Value should be done in case of Interrupt Acknowledge Signal. In order to save costs of hardware, decrementing can be performed in Stage 2. To inform this stage that decrementing has got to be done, we can add 1 extra bit to the instruction byte (most significant bit), which indicates beginning of interrupt handling. Function of operator 'Block' is therefore changed.

4. Operator 'decmux' is inserted to the schematics of Stage 1 in order to load register 'pcstore' with the value coming from the ROM Module (if there is no Interrupt Acknowledge Signal) or with decremented value coming from Stage 2 (if there is an Interrupt Acknowledge Signal). Also a connector 'pcval' is added to load this decremented value.

Schematic of the Stage 1 is shown in Appendix 2.

Since decrementing is done in Stage 2, certain changes also should be made here in such a way, that if extra bit of instruction byte is '1', Program Counter Value coming from Stage 1 must be decremented and two signals then generated: PC Load Signal and Program Flow Change Signal. *This* decremented value must be pushed onto the stack. For that reason:

1. Operator 'remb' is inserted in the design of Stage 2 in order to remove extra bit, added to the instruction byte. This is done to avoid changes in most of blocks of the stage. If interrupt handling is started, as it can be found in Tables in Appendix 1, NOP is shown to Stage 2 (because blck := $0). It means that blocks, which are

not responsible for decrementing of the PC Value and generating mentioned above signals, can receive this NOP.

2. However, 9-bit instruction is offered to the following blocks:

   -Constant generators 'alu_cntr', 'pfcch' (via operator 'con');
   -Operators 'con', 'spec_inst'.

   Therefore control specification of constant generator 'alu_cntr' is modified. This is necessary to inform 16–bit ALU of Stage 2 to decrement Program Counter Value coming from Stage 1. For decrementing we can use already created function. The only upgrading should be done in 16-bit ALU is modifying of control specification of operators 'muxl1', 'muxh', 'low_byte' inside of the schematics.

3. To load this decremented value into the Program Counter (and further push it onto the stack), Stage 2 generates PC Load Signal and Program Flow Change Signal. In operator 'spec_inst' changes are done in such a way, to generate PC Load Signal as soon as extra bit, added to the instruction byte, is '1'.

4. In the same way constant generator 'pfcch' is modified to generate PFC Signal, when '1' is recognized.

More detailed description can be found in IDaSS documentation for Stage 2. The performance of this implementation is considered on the simple program example in Appendix 3.

# Conclusions

The Interrupt Mechanism is implemented in IDaSS as it is described in the report. Now the core is able to handle seven interrupts. Interrupt Handling Mechanism is created in Stage 1. Finite State Machine of this stage is therefore changed in such a way, that in state number 1 it checks whether an interrupt happened. If so, it ignore all data received from ROM, decrement the Program Counter Value making it pointing on the same byte, and push this data onto the stack. In order to do this, functions 'one' – 'five' are modified. Also state 0 (function 'zero') is added to the design of the original Finite State Machine in order to make difference between NOP instruction from the program and NOP instruction, which is loaded in register 1 in states 1, 2 or 3, when current instruction is completed and there is no ROM Ready Signal. Some changes also made in the design of Stage 2.

The core is tested in IDaSS, show good results in pushing the correct value of the Program Counter onto the stack, and returning from the interrupt routine to the right address when RETI is encountered. However it is recommended to test this core further, interrupting various instruction, which perform operation with the Program Counter.

# References

[1] Snijders, R.A.M.J.
THE DESIGN AND IMPLEMENTATION OF A PIPELINED 8051 MICROCONTROLLER CORE.
Master's Thesis: EB 670 Eindhoven University of Technology, Section of Information and Communication Systems, 1997

[2] MICROCONTROLLER HANDBOOK.
Intel, 1983.

[3] 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

## Appendix 1 Stage 1 Operation

`State 0.` If instruction is fetched from the ROM Module and there is no ROM Ready Signal, FSM is switched to this state.

Table 1. Function **'zero'**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Reg1 | If there is no ROM Ready Signal set the value $2, to perform a **NOP**, or if there is a ROM Ready signal set the value $1 to perform a **Load** function | | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $1 to perform a **Load** function | | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0000 to perform a **Hold** function | | | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | | Set the value 1 to inform the ROM Module and 'pcstore'-register: There is a Stall | |
| Blck | Set the value $0 what means NOP for the remain of the pipeline | | | |
| Inack | Shows to the Interrupt Controller %00 – No interrupt handling | | | |
| So | If there is no ROM Ready Signal, FSM remains in this state, if there is a ROM Ready Signal, FSM is ready to read the first byte of the instruction | | | |

## State 1.

When ROM Ready Signal arises, Stage 1 receives the first byte of the instruction and according to the decoded value executes functions ‘one’ – ‘five’. Let’s consider cases.

**If the value is $0**, then current instruction – is a *single-byte instruction*, but not RET or RETI (those instructions are considered separately). For that **function ‘one’** of FSM is activated.

Table 2. Function **‘one’**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Reg1 | If there is no ROM Ready Signal, set the value $2, to perform a **NOP**, or if there is a ROM Ready signal, set the value $1 to perform a **Load** function | Set the value $4, to write into register 1 value $02 – the first byte of **LJMP**-instruction. | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $1 to perform a **Load** function | Set the value $4, which according to the control specification writes to register value $00 (the first byte of an address, loaded to PC to handle an interrupt) | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0000 to perform a **Hold** function | Set the value corresponding to the number of interrupt | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 0 to inform the ROM Module and ‘pcstore’-register: No Stalling | | Set the value 1 to shows to the ROM Module and ‘pcstore’-register: There is a Stall | |
| Blck | Set the value $1, that means the current instruction can go further to the pipeline | Set the value $0 | | |

| | | | |
|---|---|---|---|
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | Sends towards the Interrupt Controller %01 – interrupt handling | Sends towards the Interrupt Controller %00 – no interrupt handling |
| So | If there is no ROM Ready Signal set the value %000 to go to **`function 'zero'.`** If there is a ROM Ready Signal set the value %100 to remain in this state | Set the value %011 – transfers FSM to state 4 to handle the interrupt (**`function 'twelve'`**) | Set the value %100 to remain in this state |

**If the value is $1, $3 or $6**, it means that the first byte belongs to *two- or three-byte instruction*, of which the second byte has got to be placed in register 2. For that **function 'two'** of FSM is activated.

Table 3. Function **'two'**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Regl | Set the value $0 to perform **Hold** function | Set the value $4, to write into register 1 value $02 – the first byte of **LJMP**-instruction. | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $1 to perform a **Load** function | Set the value $4, to write into register 1 value $00 (the first byte of an address, loaded to PC to handle an interrupt) | Set the value $1 to perform a **Load** function | Set the value $0 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function | Set the value corresponding to the number of interrupt | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | | Set the value 1 to shows to the ROM Module and 'pcstore'-register: There is a Stall | |
| Blck | Set the value $0 | | | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | Sends towards the Interrupt Controller %01 – interrupt handling | Sends towards the Interrupt Controller %00 – no interrupt handling | |

|  |  |  |  |  |
|---|---|---|---|---|
| So | If there is no ROM Ready Signal set the value %100 to remain in this state,<br>if there is a ROM Ready Signal set the value %001 to go to the state 2 to receive the second byte of the instruction | Set the value %011 – transfers FSM to state 4 to handle the interrupt (**`function 'twelve'`**) | If there is no ROM Ready Signal set the value %100 to remain in this state, if there is a ROM Ready Signal set the value %001 to go to the state 2 to receive the second byte of the instruction | Set the value %100 to remain in this state |

**If the value is $2 or $4**, it means that the first byte belongs to *two- or three-byte instruction*, of which the second byte has got to be placed in register 3. For that **function 'three'** of FSM is activated.

Table 4. Function **'three'**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Reg1 | Set the value $0 to perform **Hold** function | Set the value $4, to write into register 1 value $02 – the first byte of **LJMP**-instruction. | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $0 to perform a **Hold** function | Set the value $4, to write into register 1 value $00 (the first byte of an address, loaded to PC to handle an interrupt) | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0001 to perform a **Load** function | Set the value corresponding to the number of interrupt | Set the value %0001 to perform a **Load** function | Set the value %0000 to perform a **Hold** function |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | | Set the value 1 to shows to the ROM Module and 'pcstore'-register: There is a Stall | |
| Blck | Set the value $0 | | | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | Sends towards the Interrupt Controller %01 – interrupt handling | Sends towards the Interrupt Controller %00 – no interrupt handling | |

| | | | | |
|---|---|---|---|---|
| So | If there is no ROM Ready Signal set the value %100 to remain in this state, if there is a ROM Ready Signal set the value %001 to go to the state 2 to receive the second byte of the instruction | Set the value %011 – transfers FSM to state 4 to handle the interrupt (**`function 'twelve'`**) | If there is no ROM Ready Signal set the value %100 to remain in this state, if there is a ROM Ready Signal set the value %001 to go to the state 2 to receive the second byte of the instruction | Set the value %100 to remain in this state |

**If the value is $5**, it means that the first byte belongs to *ACALL – instruction*. For that **function 'four'** of FSM is activated.

Table 5. Function **'four'**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Regl | Set the value $3 to write into register 1 value $01 – the first byte of **AJMP**-instruction | Set the value $4 to write into register 1 value $02 – the first byte of **LJMP**-instruction. | Set the value $3 to write into register 1 value $01 – the first byte of **AJMP**-instruction | Set the value $0 to perform a **Hold** function |
| Reg2 | Set the value $0 to perform a **Hold** function | Set the value $4, to write into register 1 value $00 (the first byte of an address, loaded to PC to handle an interrupt) | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0001 to perform a **Load** function | Set the value corresponding to the number of interrupt | Set the value %0001 to perform a **Load** function | Set the value %0000 to perform a **Hold** function |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | | Set the value 1 to shows to the ROM Module and 'pcstore'-register: There is a Stall | |
| Blck | Set the value $0 | | | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | Sends towards the Interrupt Controller %01 – interrupt handling | Sends towards the Interrupt Controller %00 – no interrupt handling | |

| | | | | |
|---|---|---|---|---|
| So | If there is no ROM Ready Signal set the value %100 to remain in this state,<br>if there is a ROM Ready Signal set the value %011 to switch the FSM to state 4, which force two Pushes after a forced Jump. **`Function`** **`'twelve'`** is going to be executed. | Set the value %011 – transfers FSM to state 4 to handle the interrupt (**`function`** **`'twelve'`**) | If there is no ROM Ready Signal set the value %100 to remain in this state, if there is a ROM Ready Signal set the value %001 to go to the state 2 to receive the second byte of the instruction | Set the value %100 to remain in this state |

**If the value is $7**, it means that the first byte belongs to *RET* or *RETI – instruction.* For that **function 'five'** of FSM is activated (as written before, an interrupt request should not be responded to after RETI. That is why those instructions are considered in the report separately. The difference between those instructions is taken out from the bit 4)

Table 6a. Function **'five' (RET)**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Reg1 | Set the value $6 to write into the register value $D0 – the first byte of **Pop** instruction | Set the value $4 to write into register 1 value $02 – the first byte of **LJMP**-instruction | Set the value $6 to write into the register value $D0 – the first byte of **Pop** instruction | |
| Reg2 | Set the value $3 to write into the register value $EA (Pop of the high-order byte of the PC Value) | Set the value $4 to write into register 1 value $00 (the first byte of an address, loaded to PC to handle an interrupt) | Set the value $3 to write into the register value $EA (Pop of the high-order byte of the PC Value) | |
| Reg3 | Set the value %0000 to perform a **Hold** function | Set the value corresponding to the number of interrupt | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 1 to inform the ROM Module and 'pcstore'-register: There is a Stall | Set the value 0 to inform the ROM Module and 'pcstore'-register: no Stall | Set the value 1 to shows to the ROM Module and 'pcstore'-register: There is a Stall | Set the value 0 to inform the ROM Module and 'pcstore'-register: no Stall |
| Blck | Set the value $0 | | | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | Sends towards the Interrupt Controller %01 – interrupt handling | Sends towards the Interrupt Controller %00 – no interrupt handling | |
| So | Set the value %010 to switch FSM to state 3, where it performs a **forced pop** (**`function 'eleven'`**). | Set the value %011 – transfers FSM to state 4 to handle the interrupt (**`function 'twelve'`**) | Set the value %010 to switch FSM to state 3, where it performs a **forced pop** (**`function 'eleven'`**). | Set the value %100 to remain in this state |

Table 6b. Function **'five'** **(RETI)**

| | NO STALL | | STALL | |
|---|---|---|---|---|
| | No interrupt | Interrupt | No interrupt | Interrupt |
| Regl | Set the value $6 to write into the register value $D0 – the first byte of **Pop** instruction | | | |
| Reg2 | Set the value $3 to write into the register value $EA (Pop of the high-order byte of the PC Value) | | | |
| Reg3 | Set the value %0000 to perform a **Hold** function | | | |
| St | Set the value 1 to inform the ROM Module and 'pcstore'- register: There is a Stall | | | |
| Blck | Set the value $0 | | | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | | | |
| So | Set the value %010 to switch FSM to state 3, where it performs a **forced pop** (`function 'eleven'`). | | | |

After the first byte is received and decoded, Stage 1 is ready to receive the second byte of an instruction (if it is available). For that reason State Machine is now in state number 2.

## State 2.

**If the value is $1 or $2**, it means that current instruction is a *two-byte instruction*, but not ACALL instruction. For that **function 'six'** of FSM is activated.

Table 7. Function **'six'**

| | NO STALL | STALL |
|---|---|---|
| Reg1 | If there is no ROM Ready Signal, set the value $2 to perform a **NOP**, or if there is a ROM Ready signal set the value $1 to perform a **Load** function | Set the value $0 to perform a **Hold** function |
| Reg2 | Set the value $1 to perform a **Load** function | Set the value $0 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | Set the value 1 to inform the ROM Module and 'pcstore'-register: There is a Stalling |
| Blck | Set the value $1, that means the current instruction (which is a two-byte instruction) can go further to the pipeline | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | |
| So | If there is no ROM Ready Signal set the value %000 to go to **`function 'zero'`**. If there is a ROM Ready Signal set the value %100 to switch FSM to the state 1 where the first byte of the next instruction is received. | Set the value %001 to remain in this state. |

**If this value $3**, it means that the second byte belongs to the *three-byte instruction*, of which the third byte has got to be placed in register 3.

Table 8. Function **'seven'**

| | NO STALL | STALL |
|---|---|---|
| Reg1 | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0001 to perform a **Load** function | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'- register: No Stalling (current instruction is not completed) | |
| Blck | Set the value $0 | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | |

|  |  |
|---|---|
| So | If there is no ROM Ready Signal set the value %001 to remain in this state.<br>If there is a ROM Ready Signal set the value %010 to switch FSM to the state 3 where the third byte of the next instruction is received. |

**If this value $4**, it means that the second byte belongs to the *three-byte instruction*, of which the third byte has got to be placed in register 2. For that **function 'eight'** of FSM is activated.

Table 9. Function **'eight'**

|  | NO STALL | STALL |
|---|---|---|
| Regl | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $1 to perform a **Load** function | |
| Reg3 | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'- register: No Stalling (current instruction is not completed) | |
| Blck | Set the value $0 | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | |
| So | If there is no ROM Ready Signal set the value %001 to remain in this state.<br>If there is a ROM Ready Signal set the value %010 to switch FSM to the state 3 where the third byte of the next instruction is received. | |

**If the value is $6**, it means that the second byte belongs to *LCALL – instruction.* For that **function 'nine'** of FSM is activated.

Table 10. Function **'nine'**

|  | NO STALL | STALL |
|---|---|---|
| Regl | If there is no ROM Ready Signal set the value $0 to perform a **Hold** function,<br>If there is a ROM Ready Signal set the value $4 to write into register 1 value $02 – the first byte of **LJMP**- instruction | |
| Reg2 | Set the value $0 to perform a **Hold** function | |
| Reg3 | Set the value %0001 to perform a **Load** function (the instruction byte is copied into the third register to simplify target address calculation | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'- register: No Stalling (current instruction is not completed) | |
| Blck | Set the value $0 | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | |
| So | If there is no ROM Ready Signal set the value %001 to remain in this state.<br>If there is a ROM Ready Signal set the value %011 to switch FSM to the state where **`function 'twelve'`** is executed (**Push** onto the stack the low-order byte) | |

After the second byte is received and decoded, Stage 1 is ready to receive the third byte of an instruction (if it is available). For that reason State Machine is now in state number 3. In this state also a 'forced pop' is performed.

## State 3.

If the value is **$3 or $4**, it means that current instruction is any of *three-byte instructions*. For that **function 'ten'** of FSM is activated.

Table 11. Function **'ten'**

| | NO STALL | STALL |
|---|---|---|
| Reg1 | If there is no ROM Ready Signal, set the value $2 to perform a **NOP**, or if there is a ROM Ready signal set the value $1 to perform a **Load** function | Set the value $0 to perform a **Hold** function |
| Reg2 | Set the value $1 to perform a **Load** function | Set the value $0 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function | |
| St | Set the value 0 to inform the ROM Module and 'pcstore'-register: No Stalling | Set the value 1 to inform the ROM Module and 'pcstore'- register: There is a Stalling |
| Blck | Set the value $1, that means the current instruction (which is a three-byte instruction) can go further to the pipeline | |
| Inack | Sends towards the Interrupt Controller %00 – no interrupt handling | |
| So | If there is no ROM Ready Signal set the value %000 to go to **`function 'zero'`**. If there is a ROM Ready Signal set the value %100 to switch FSM to the state 1 where the first byte of the next instruction is received. | Set the value %010 to remain in this state. |

**If this value $1**, it means that a 'forced pop' will be executed. For that **function 'eleven'** of FSM is activated.

Table 12. Function **'eleven'**

| | NO STALL | STALL |
|---|---|---|
| Reg1 | Set the value $0 to perform a **Hold** function | |
| Reg2 | Set the value $2, which according to the control specification writes to register value $E9 to pop low-order byte of the PC Value | Set the value $00 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function. | |
| St | Set the value $1 to inform the ROM Module and 'pcstore'- register: There is a Stalling. | |

| | | |
|---|---|---|
| Blck | Set the value $1 | |
| Inack | Sends towards the Interrupt Controller value %00 – No interrupt handling | |
| So | Set the value %001 to transfer FSM to state 2 | Set the value %010 to remain in this state till Stall is deactivated |

## State 4.

The fourth state of the Finite State Machine is inserted to make it possible to force two **Pushes** after a forced **Jump**. As written before this is done to replace a **Call** with a **Jump** and two **Pushes**. However this state can be considered as two states. That is performed by **functions 'twelve' and 'thirteen'**.

Table 13. Function **'twelve'**

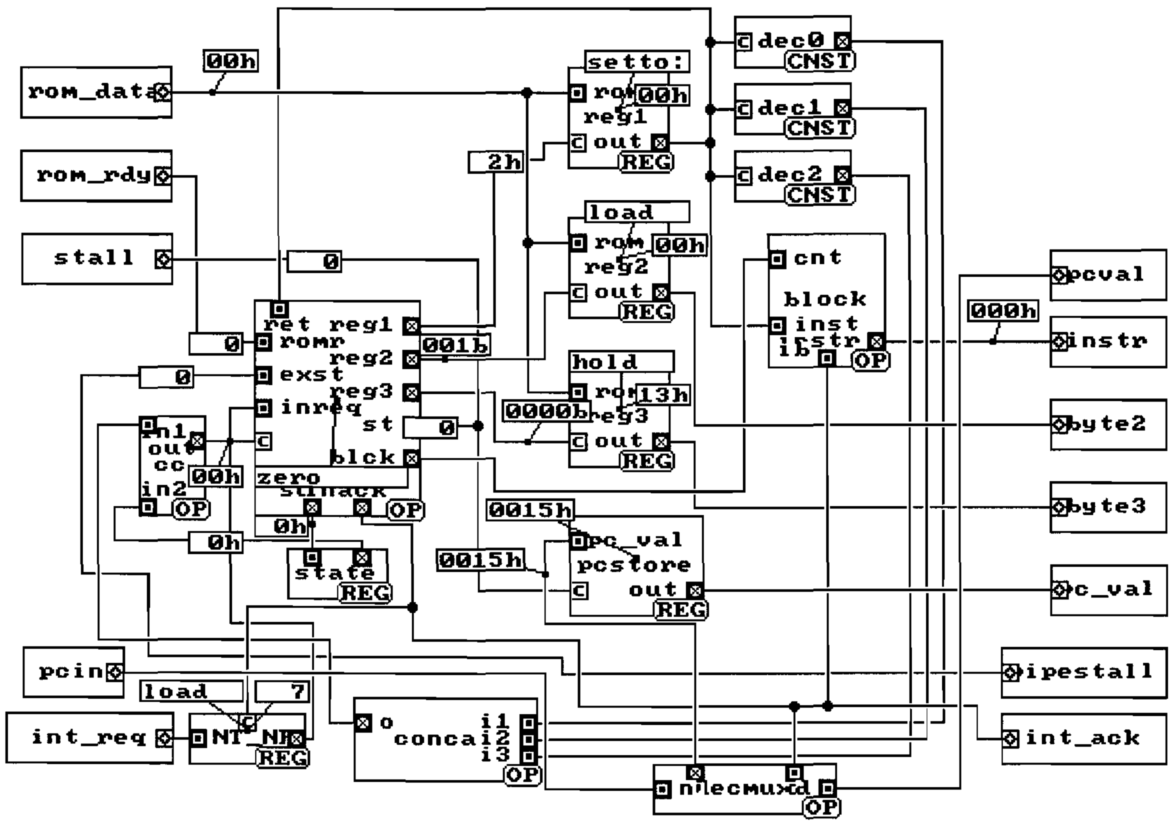
| | NO STALL | STALL |
|---|---|---|
| Reg1 | Set the $5, which according to the control specification writes to register value $C0 – the first byte of **Push** instruction (first **Push**) | Set the value $0 to perform a **Hold** function |
| Reg2 | Set the value $2, which according to the control specification writes to register value $E9 to push low-order byte of the PC Value | Set the value $00 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function. | |
| St | Set the value $1 to inform the ROM Module and 'pcstore'- register: There is a Stalling. | |
| Blck | Set the value $1 | |
| Inack | Sends towards the Interrupt Controller value %00 – no interrupt handling | |
| So | Set the value %111 to transfer FSM to execute **function 'thirteen'** | Set the value %011 to remain in this state till Stall is deactivated |

Table 14. Function **'thirteen'**

| | NO STALL | STALL |
|---|---|---|
| Reg1 | Set the $0 to perform a **Hold** function | |
| Reg2 | Set the value $3, which according to the control specification writes to register value $EA to push high-order byte of the PC Value | Set the value $00 to perform a **Hold** function |
| Reg3 | Set the value %0000 to perform a **Hold** function. | |
| St | Set the value $1 to inform the ROM Module and 'pcstore'- register: There is a Stalling. | |
| Blck | Set the value $1 | |
| Inack | Sends towards the Interrupt Controller value %00 – no interrupt handling | |

| | | |
|---|---|---|
| So | Set the value %001 to transfer FSM to state 2 | Set the value%111 to remain in this state till Stall is deactivated |

# Appendix 2 Stage 1 Schematics



# Appendix 3 Example

Here we will consider the performance of the core with implemented Interrupt Mechanism. For example let's assume the text of the program as follows (from IDaSS):

| | | | | | | |
|---|---|---|---|---|---|---|
| 0000h | 00 | 01 | 00 | 00 | 00 | 00 |
| 0006h | 00 | 00 | 00 | 00 | 00 | 00 |
| 000Ch | 00 | 00 | 32 | 00 | 00 | 00 |
| 0012h | 00 | 00 | | | | |

| Clock Cycles | Operation |
|---|---|
| 1 | Interrupt Request |
| 3 | Interrupt Acknowledge |
| 8 | RETI received |
| 9 | **Push** of low-byte (00) |
| 10 | **Push** of high-byte (00) |
| 14 | Continue executing the program at address 0000h |

| Clock Cycles | Operation |
|---|---|
| 2 | Interrupt Request |
| 3 | Interrupt Acknowledge |
| 8 | RETI received |
| 9 | **Push** of low-byte (00) |
| 10 | **Push** of high-byte (00) |
| 14 | Continue executing the program at address 0000h |

| Clock Cycles | Operation |
|---|---|
| 3 | Interrupt Request |
| 4 | Interrupt Acknowledge |
| 9 | RETI received |
| 10 | **Push** of low-byte (01) |
| 11 | **Push** of high-byte (00) |
| 15 | Continue executing the program at address 0001h |

| Clock Cycles | Operation |
|---|---|
| 4 | Interrupt Request |
| 8 | Interrupt Acknowledge |
| 13 | RETI received |
| 14 | **Push** of low-byte (00) |
| 15 | **Push** of high-byte (00) |
| 19 | Continue executing the program at address 0000h |