Chapter 6: Implementation of Control Part

6.1 Introduction

After having analyzed the application and having specified the datapath (execution unit), and having decided on instructions in the instruction set, it is now the turn to design the control part of the processor that will help implement the functionality of the instructions on the datapath (execution unit) by generating the appropriate sequences of control signals that control the data transfers and functions of the various functional units in the datapath (execution unit).

Given that several functional units in the datapath (execution unit) have been designed as multicycle sequential computing units with their own dedicated controllers that can be triggered by a higher level controller, it was considered appropriate that a microprogrammed control unit with dynamically programmable micro cycle time (e.g. microinstruction specifiable number of wait cycles – after which the next microinstruction should be fetched) would be an efficient way of implementing the control part. A microinstruction that triggers a sequential functional unit in the datapath can also specify the number of wait clock cycles (that the triggered functional unit would require to generate the results) after which the next microinstruction would be executed so that it can use the results that have been computed by its preceding microinstruction. This approach effectively implements a dynamically programmable micro cycle time and helps the microprogrammer to optimize his microprograms for implementing various instructions – without burdening him with the explicit implementation of wait micro cycles through a hardware.

With the above stated philosophy, the technique of hardware flow-charting [32] has been used to develop the microprograms (microinstructions/control word sequences) required to implement the functionality of each instruction on the datapath (execution unit) micro-architecture already specified in chapter 5.
6.2 Hardware Flowchart Methodology

Hardware flowcharts capture the activities inside a processor on a control-step-by-control-step basis [32]. In the hardware flowcharts, register transfer notation is used to describe the operations in the datapath (execution unit) and they capture the essence of what the processor is doing in a step-by-step (state-by-state) manner. They show sequential state flows implementing associated concurrent tasks where each task is a register transfer task. For example, Reg $\rightarrow$ Bus $\rightarrow$ Input Register of functional unit.

The above task is interpreted as “transfer the contents of register Reg via the Bus to the input register of the functional unit”.

The hardware flowcharts serve both as a design representation at the Register Transfer (RT) level and as a tool for optimization of the design at the RT-level iteratively.

6.2.1 Level-I and Level-II Hardware Flowcharts

There are two kinds of RT-level tasks that each instruction execution accomplishes. One kind of tasks are the computational tasks unique to each individual instruction and are called operational tasks. The second kind of tasks are the tasks that are performed by all the instructions (and are, therefore, common to all the instructions) and are called housekeeping tasks. The housekeeping tasks include incrementing the Program Counter (PC), fetching the next instruction from the memory using PC as the address and loading it into the instruction register of the processor.

In the level-I flowcharts, one records these two kinds of tasks separately in two separate juxtaposed columns – one showing the operational tasks and the other showing the housekeeping tasks. The tasks are arranged in boxes in each column.
Figure 6.1: Level-I Flowchart for ADDII Instruction

Figure 6.1 shows the level-I flowchart sequences of the ADDII (add two integer operands) instruction. Each box represents a state and each line entry in a box is a task. The tasks are usually expressed in the register-transfer notation i.e. source-bus-destination format.

The tasks under each column which can be accomplished concurrently within the execution unit (because their data inputs are available, and execution unit resources and rules of operation permit it) should be placed in a single box under that column. The tasks, which cannot be accomplished concurrently, must be put in the next or a subsequent box under that column.

In figure 6.1, under column-1 in box-1, as a first task, the Ptr1_field_of_Instruction Register (IR) is communicated to the integer memory (Mint) as an address to read the first operand. Since this field also acts as the destination pointer, therefore, it needs to be saved in Ptr1_reg. In box-2, the first task transfers the contents of integer memory (Mint) output register (16-bit) to first input register i.e. In_reg1_IASC of IASC (Integer Add Sub Compare) functional block via the bus. The second task in box-2 communicates the Ptr2_field_of_IR as the address to integer memory (Mint) to read the second operand. In box-3 under column-1, the contents of integer memory output register (second operand) are transferred to input register In_reg2_IASC of IASC block via the
bus. In box-4 under column-1, there is an empty state because one state is needed to compute the result by the IASC functional block. In box-5 under column-1, the result from the output register of IASC block is transferred to integer memory (Mint) at the location pointed to by Ptr1_reg via the bus.

Under column-2 are the housekeeping tasks. In box-1, Program Counter (PC) is incremented. In box-2, PC value is sent as the address to Program Memory (PM) to fetch the next instruction (in our processor the program memory output register also serves as the instruction register).

6.2.2 Level-I to level-II Hardware Flowchart Conversion

These two kinds of tasks (operational and housekeeping) are independent of each other but have their individual sequential orders demarcated by boxes. One easy (but not at all efficient) way of executing an instruction is for the controller to go through the successive control steps that make the execution unit do the tasks in the successive boxes under the column-1 and then under the column-2. This way, the number of states required to accomplish the execution of the instruction would be equal to the sum of number of boxes under column-1 and column-2. This is not very efficient and one would seek ways to reduce the number of control steps needed to execute an instruction. This can be accomplished by merging the tasks under column-2 (the housekeeping tasks) with the tasks under column-1 if the datapath (execution unit) architecture and rules of operation permit it.

The rules of merging [32] are:

- Individual tasks in the first box under column-2 can be merged with the tasks in a single box under column-1 or more than one box under column-1 depending upon whether datapath (execution unit) architecture and the associated rules of operation permit the merger.
- Tasks from the second/next box under column-2 should be merged with tasks in box (es) under column-1 only at the box (es) following the last box number into which any task of the first/previous box under column-2 was merged.
- Direction of merging is always from column-2 into column-1.
- The first box under column-2 whose full set of tasks cannot be merged into boxes under column-1 as well as subsequent boxes under column-2 are appended as additional boxes after the last box under column-1.

The above rules are designed to preserve the sequential order of tasks under column-1 even after their merger with tasks in column-2.

The merged flowcharts so produced would typically (but not necessarily) have lesser number of boxes as compared to the sum of number of boxes under column-1 and column-2. A reduced number of boxes imply a reduction in the number of control steps (through better utilization of capabilities available within the execution unit). Such merged flowcharts with some additional information annotated in them are called level-II flowcharts.

The annotation of certain information items to level-II flowcharts makes the transition from flowcharts to hardware easier. Figure 6.2 shows the format for a level-II flowchart state which has more boxes to annotate/capture the details associated with a state – to ease the synthesis of logic of the control part.

<table>
<thead>
<tr>
<th>Register Transfers and Operations</th>
<th>Access Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation type</td>
<td></td>
</tr>
<tr>
<td>State ID</td>
<td>Next State</td>
</tr>
<tr>
<td>Next State</td>
<td>Next State</td>
</tr>
<tr>
<td>Transition</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.2: Annotated Flowchart Template**

The information annotated in figure 6.2 is:

- **Access Type**: This box is used to annotate the type of memory access carried out during the state. We use the following codes for this annotation:
  - **DR**: Data Read (read-in of data from the data memory).
- **DS**: Data Source (sending the data from memory output register to the functional unit input register via bus).
- **DW**: Data Write (writing data into the data memory).
- **IR**: Instruction Read.

- **Operation Type**: This field is used to annotate the type of operation performed by the execution unit. For example: addition, subtraction, multiplication, division, PC incrementing etc.

- **Next State Transition Type**: This field is used to annotate the branching type to select the next state. We use the following codes for different branch types:
  - **IB** (Instruction Branch): This is used by the last state of an instruction’s flowchart to indicate that the next state would be the first state of the next instruction – which is generated by the instruction decoder.
  - **WB** (Wait Branch): This is used to inject wait microcycles.
  - **DB** (Direct Branch): Where a state provides the state name of its successor state.
  - **CB** (Conditional Branch): Where the next state address is provided by BCON block (e.g. branch control unit).

- **State ID**: It contains an identifier (name) for the present state.

- **Next state**: contains an identifier (name) for the next state in the case of direct branch/delayed direct branch (after a specified wait period to achieve dynamically programmable microcycle time).

This method of hardware flowcharting has been used to derive the control steps needed for the execution of each of the instructions. These flowcharts have been used directly to build the control sequences for the implementation of the instructions.
To write the hardware flowchart for any instruction, we need to have the details of
the micro-architecture of the datapath (execution unit) in the processor and its associated
rules of operation besides a clear understanding of the semantics as well as the format of
the instruction.

As an example the level-II hardware flowchart for the ADDII instruction for the
datapath (execution unit) micro-architecture described in figure 6.1 in section 6.2.1 is
given below. Hardware flowcharts also take into account a set of rules of operation
associated with the operations of the various functional blocks/units in the execution unit.
Rules of operation for our datapath (execution unit) are given below.

6.2.3 Rules of operation

- Only port-A of the dual-ported integer memory (Mint) is used for data
  transfers from and to system registers (except output to ADC which is carried
  out via port-B dedicated for the purpose).
- Memory read i.e. transfer of data from the location whose address is input to
  the memory on port-A/port-B to the memory’s data output register is a single
  cycle operation.
- Transfer of data from the memory data output register to any functional block
  input register (whether via data bus or through a direct path) is also a single
  cycle operation.
- Memory write i.e. transfer of data from any functional block output register to
  a memory location via port-A/port-B is also a single cycle operation.
- There is a direct path from instruction register pointer fields via DMAU to
  integer and floating-point memory address ports.
- There is a single 32-bit data bus connecting all functional block input/output
  registers and memories that can support 16-bit or 32-bit transfers.
- In case of single operand functional unit, triggering of the functional block is
done together with the loading of operand.
- In case of dual operand functional unit, triggering of the functional block is
done together with the loading of second operand.
The output of IASC functional block is available after 1 clock cycle of loading the operands.

The output of FASC functional block is available after 4 clock cycles of loading the operands.

The output of FMPY functional block is available after 6 clock cycles of loading the operands.

The output of FDIV functional block is available after 26 clock cycles of loading the operands.

The output of IGEN functional block is available after 14 clock cycles of its triggering.

The output of SEXP functional block is available after 54 clock cycles of loading the operand.

The output of SCOS functional block is available after 78 clock cycles of loading the operand.

The output of ITOF functional block is available after 1 clock cycle of loading the operand.

The output of F2IS functional block is available after 1 clock cycle of loading the operand.

The output of STOF functional block is available after 1 clock cycle of loading the operand.

All the constants that are used in the implementation of the instruction are stored in special registers (which are part of a set of temporary registers).

Based on the execution unit description (given in section 5.2 of chapter 5) and the rules of operation described above, an example second-level flowchart for instruction ADDII is given below:

**ADDII: Format**

<table>
<thead>
<tr>
<th>000000</th>
<th>Destination Ptr1 (10)</th>
<th>Source Ptr2 (10)</th>
<th>000000 xxxxxx</th>
</tr>
</thead>
</table>

98
ADDII: Semantic Description
Add two integer data pointed to by ptr1 and ptr2 respectively in the integer memory and store the result in the integer memory at location pointed to by ptr1.

ADDII: Description of Level-II flowcharts

State Addii-1: In this state, operand-1 is read from the memory. To achieve this, address from pointer-1 field of instruction register is communicated to integer memory (via DMAU which has a direct path from its output to integer memory address input). As according to our instruction format (two operand format) the result is to be stored at the location of the first operand, therefore it is required to store this address into a register, which is the Ptr1_reg. Now, ADDII is a single word instruction and, so the PC value is needed to be incremented by one to point the next instruction. Only these two tasks can be performed in this state, and we take a direct branch to the next state (Addii-2). State Addii-1 is shown in figure 6.3.

State Addii-2: In this state, operand-1 available in the memory output register is loaded into the first input register In_reg1_IASC of the IASC block (Integer Add Subtract Compare functional block) via the bus. Simultaneously data from the location indicated by Ptr2_field_of_IR is read into the memory output register of the integer memory (Mint). Only these two tasks can be performed in this state, and we take a direct branch to the next state (Addii-3). State Addii-2 is shown in figure 6.4.
State ADDii-3: In this state, operand-2 available in the memory output register is loaded into the second input register In_reg2_IASC of the functional block (IASC block) via the bus. Here we have captured the type of operation (e.g. ADD) that is performed in the IASC unit. Only these tasks are performed in this state, and we take a direct branch to the next state (Addii-4). State Addii-3 is shown in figure 6.5.

State Addii-4: In this state, IASC functional block computes the result and saves it in its result register. In this state one sends the contents of PC as address to PM (Program Memory) to fetch the next instruction. Direct branch is taken to the next state Addii-5. State Addii-4 is shown in figure 6.6.
State Addii-5: In this state, contents of the result register RO_reg_IASC of IASC block are stored in the integer memory at the memory location pointed to by Ptr1_reg and an instruction branch is taken (which means that the next state address is taken from the instruction decoder). State Addii-5 is shown in figure 6.7.

![Figure 6.7: Hardware Flowchart for State Addii-5 instruction](image)

The complete level-II flowchart for instruction ADDII is shown in figure 6.8.

![Figure 6.8: Hardware Flowchart for ADDII Instruction](image)
Here is another example of the hardware flowcharts of an application specific instruction i.e. EXP instruction (which computes the value of the exponential function):

**EXP: Format**

<table>
<thead>
<tr>
<th></th>
<th>Destination Ptr1</th>
<th>Source Ptr2</th>
<th>xxxxxxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>110000</td>
<td>(6)</td>
<td>(10)</td>
<td>(6)</td>
</tr>
</tbody>
</table>

**EXP: Semantic Description**

This instruction computes the value of the exponential function. The Ptr2 field of instruction specifies the argument location of the function in the floating-point memory. The computed function value is stored at the Ptr1 location in the floating-point memory.

**EXP: Description of level-II flowchart states**

**State Exp-1:** In this state, operand (whose address is given by source Ptr2 field of the instruction) is read from the floating-point memory (MFP). To achieve this, address from pointer-2 field of instruction register is given to floating-point memory. According to the instruction format, result is to be stored at the pointer-1 field of instruction register, so it is required to store this address into a register, which is the Ptr1_reg. Only these two tasks are performed in this state, and direct branch is taken to the next state Exp-2. State Exp-1 is shown in figure 6.9.

![Figure 6.9: Hardware Flowchart for State Exp-1](image-url)
**State Exp-2:** In this state, operand available in the MFP output register is loaded into the input register of the functional block F2IS (Floating-point format To Integer and Special format converter) via the bus to convert the floating-point operand into a 25-bit special 2's complement format (special 25-bit 2’s complement format with implied binary point after the two most significant bits), which is used in SEXP (which computes the value of the exponential function) functional block as input.

Only this task is performed in this state and direct branch is taken to the next state Exp-3. State Exp-2 is shown in figure 6.10.

![Figure 6.10: Hardware Flowchart for State Exp-2](image)

**State Exp-3:** In this state, the PC value is incremented by one to point to the next instruction. Since F2IS functional block takes 1 clock cycle to compute the result (this task is not listed in the flowchart state as it is performed by the dedicated control embedded in the F2IS block), therefore, only this task is performed in this state and direct branch is taken to the next state Exp-4. State Exp-3 is shown in figure 6.11.

![Figure 6.11: Hardware Flowchart for State Exp-3](image)

**State Exp-4:** In this state, the result from the F2IS functional block is loaded into the input register of the SEXP functional block. The result is available after 54 clock
cycles of loading the input. This wait count (54 clock cycles) is loaded into the wait counter, which is a down counter and the next address is stored in the Saved_NA_reg register. It then goes into the wait state. It continues to remain in the wait state until the down counter becomes zero. Then, the saved_next_address becomes the next_address. State Exp-4 is shown in figure 6.12.

State Exp-4: In this state, contents of the result register RO_reg_Exp of SEXP functional block is loaded into the input register of the STOF functional block to convert the 25-bit special 2’s complement format (special 25-bit 2’s complement format with implied binary point after the two most significant bits) data into the floating-point data. STOF functional block takes 1 clock cycle to compute the result. Only this task is performed in this state and direct branch is taken to the next state Exp-6. State Exp-5 is shown in figure 6.13.

State Exp-5: In this state, PC is sent as address to PM (Program Memory) to fetch the next instruction. Only this task is performed in this state direct branch is taken to the next state Exp-7. State Exp-6 is shown in figure 6.14.
State Exp-7: In this state, contents of the result register RO_reg_STOF of STOF functional block are stored in the floating-point memory at the memory location pointed to by Ptr1_reg and an instruction branch is taken (which means that the next state address is taken from the instruction decoder). State Exp-7 is shown in figure 6.15.

The complete level-II flowchart for instruction EXP is shown in figure 6.16.
Figure 6.16: Hardware Flowchart for Exp Instruction
6.3 Micro-instruction Design Philosophy

After writing the hardware flowcharts for all the instruction, the next step is to develop the control words (micro-instructions) for all the instructions. A control word (micro-instruction) consists of various control fields that control the operations of the ASIP during a microcycle by passing controlling signals in encoded form to its respective units. A horizontal micro-coding strategy has been used to define the number and bit-widths of different fields in the micro-instruction as described below.

6.3.1 Control Word (Micro-instruction) Description

Horizontal micro-coding dedicates one control field per functional unit in the micro-architecture. The bit-width of the control field associated with a functional unit depends upon the number of distinct functions that the functional unit has been designed to perform. Null function is also to be counted as one of the valid functions. Following this philosophy, we have divided the control words (micro-instructions) into seventeen (17) fields of requisite bit widths (depending upon the distinct functions that the functional unit performs). This result in a 60-bit micro-instruction. Control ROM output register crom_reg holds the control word (micro-instruction) during a microcycle in which the control word controls the operations of the ASIP (both execution unit and the control part). Details of the control word fields and their encoding is shown below:

Field1: crom_reg (59 downto 58): Program Memory Access Control (2-bits)

"00" : NOP (No operation).
"01" : Read instruction from program memory.
"10" : Source Instruction Register (IR) half word (15 downto 0) onto bus(15 downto 0).
"11" : Source Instruction Register (IR) full word (31 downto 0) onto bus(31 downto 0).

Field2: crom_reg(57downto55): Program Counter Logic Unit (PCLU) Control (3-bits)

"000" : NOP.
"001" : PC <= PC + 1.
"010": PC <= absolute value (from bus).
"011": PC <= initialized value (from Instruction Register).
"100": Bus <= PC.
"101": NOP.
"110": NOP.
"111": NOP.

Field3: crom_reg (54 downto 48): Data Memory Control (7-bits)

Sub-field1: crom_reg (54 downto 53): Memory Select (2-bits)

"00": NOP.
"01": enable integer memory.
"10": enable floating-point memory.
"11": NOP.

Sub-field2: crom_reg (52 downto 51): Read Write Control (2-bits)

"00": Data Read (read-in of data from the data memory).
"01": Data Write (writing data into the data memory).
"10": Data Source (sending the data from memory output register to the functional unit input register via bus).
"11": Both Data Read and Data Source.

Sub-field3: crom_reg (50 downto 48): Data Memory Address Selection (3-bits)

"000": NOP.
"001": Instruction pointer1 (ptr1_field) of Instruction register (IR).
"010": Instruction pointer2 (ptr2_field) of Instruction register (IR).
"011": Stored Pointer1 (ptr1_reg).
"100": Stored Pointer2 (ptr2_reg).
"101": Speech sample queue pointer for storing computed speech samples (Sq_reg).
"110": Loop stack pointer (Sp_reg).
"111": NOP.

Field4: crom_reg (47 downto 43): Data Memory Address Unit (DMAU) Control (5-bits)

"00000": NOP.
"00001": ptr1_reg <- ptr1_field of IR.
"00010": ptr1_reg <- ptr1_reg + 1.
"00011": ptr1_reg <- ptr1_reg - 1.
"00100": ptr1_reg <- ptr1_reg + offset value.
"00101" : ptr1_reg <- Bus (9 downto 0).
"00110" : ptr2_reg <- ptr2_field of IR.
"00111" : ptr2_reg <- ptr2_reg + 1.
"01000" : ptr2_reg <- ptr2_reg - 1.
"01001" : ptr2_reg <- ptr2_reg + offset value.
"01010" : ptr2_reg <- Bus (9 downto 0).
"01011" : Sp_reg <- ptr1_field of IR.
"01100" : Sp_reg <- Sp_reg + 1.
"01101" : Sp_reg <- Sp_reg - 1.
"01110" : Sq_reg <- Buffer1_ptr_reg.
"01111" : Sq_reg <- Sq_reg + 1.
"10000" : Buffer1_ptr_reg <- ptr1_field of IR.
"10001" : Buffer1_ptr_reg <-> Buffer2_ptr_reg (For swapping).
"10010" : Buffer2_ptr_reg <- ptr2_field of IR.
"Rest" : NOP.

Field6: crom_reg (40 downto 37): Functional unit to Bus Transfer Control (F2BC) (4-bits)

"0000" : NOP.
"0001" : NOP.
"0010" : NOP.
"0011" : Bus <- IASC (15 downto 0).
"0100" : Bus <- FASC (31 downto 0).
"0101" : Bus <- FMPY (31 downto 0).
"0110" : Bus <- FDIV (31 downto 0).
"0111" : Bus <- SCOS1 (sine value) (24 downto 0).
"1000" : Bus <- SCOS2 (cos value) (24 downto 0).
"1001" : Bus <- SEXP (24 downto 0).
"1010" : Bus <- IGEN (19 downto 0).
"1011" : Bus <- ITOF (31 downto 0).
"1100" : Bus <- F2IS (15 downto 0).
"1101" : Bus <- F2IS (24 downto 0).
"1110" : Bus <- STOF (31 downto 0).
"1111" : Bus <- IPIF (15 downto 0).

Field7: crom_reg(36 downto 33): Bus To Functional Unit Transfer Control (b2fc) (4-bits)

"0000" : NOP.
"0001" : FASC-enable input_reg1.
"0010" : FASC-enable input_reg2, ADD, Start.
"0011" : FASC-enable input_reg2, SUB, Start.
"0100" : FMPY-enable input_reg1.
"0101" : FMPY-enable input_reg2, Start.
"0110" : FDIV-enable input_reg1.
"0111" : FDIV-enable input_reg2, Start.
"1000" : SCOS-enable input_reg, Start.
"1001" : SEXP-enable input_reg, Start.
"1010" : ITOF-enable input_reg, Start (16-bit data).
"1011" : ITOF-enable input_reg, Start (20-bit data).
"1100" or "1110" : F2IS-enable input_reg, Start.
"1101" : STOF-enable input_reg, Start.
"1111" : IGEN-Start.

**Field8 : crom_reg (32 downto 26): Integer Unit Control (7-bits)**

**Sub-field1: crom_reg (32 downto 31): Enable input registers (2-bits)**

"00" : NOP.
"01" : IASC-enable input_reg1.
"10" : IASC-enable input_reg2, trigger add operation.
"11" : IASC-enable input_reg2, trigger subtract operation.

**Sub-field2: crom_reg (30 downto 28): Branch Condition (3-bits)**

Set IASC_flag register to ‘1’ if associated condition is true otherwise resets it to ‘0’. NOP leaves the flag register unaltered.

"000" : NOP.
"001" : A /= B.
"010" : A < B.
"011" : A <= B.
"100" : A > B.
"101" : A >=B.
"110" : NOP.
"111" : A = B.

**Sub-field3: crom_reg (27): Enable bit for storing branch condition from instruction register (1-bit)**

'0' : Storing not enabled.
'1' : Storing enabled.

**Sub-field4: crom_reg (26): Enable bit for selecting branch condition (1-bit)**

'0' : Select branch condition from the Control word.
'1' : Select branch condition from the Instruction register.
Field 9: \texttt{crom\_reg (25)}: Start\_IPIF Control (1-bit)

'S0': NOP.
'S1': Start input interface control.

Field 10: \texttt{crom\_reg (24 downto 22)}: OUTput Controller Control (OUTC) (3-bits)

'S000': NOP.
'S001': Load the number of samples to be outputted into the nspfr\_reg.
'S010': Load the number of clock cycles that the system needs to wait in between the present output sample and the next output sample into the Ts\_reg.
'S011': Load the starting address of the memory location in the integer memory (where computed speech samples are stored) into the output\_buffer\_pointer\_reg.
'S100': Enable start signal to start the output controller.
'S101': NOP.
'S110': NOP.
'S111': Reset output controller.

Field 11: \texttt{crom\_reg (21 downto 14)}: Next Address Field (8-bits)

This 8-bit field gives the address of the next control word to be fetched from control store in case of a direct branch (DB).

Field 12: \texttt{crom\_reg (13 downto 12)}: Next Address Control (2-bits)

'S00': Direct Branch (DB).
'S01': Instruction Branch (IB).
'S10': Conditional Branch (CB) on the basis of Control Branch Type field.
'S11': Wait Branch (WB), wait for the wait delay to be over before going to the next state.

Field 13: \texttt{crom\_reg (11 downto 10)}: Control Branch Type (2-bits)

'S00': Branching is done on the basis of IASC\_flag from the IASC block.
'S01': Branching is done on the basis of FASC\_flag from the FASC block.
'S10': Branching is done on the basis of IPIF\_ack flag from the IPIF block.
'S11': Branching is done on the basis of end\_of\_frame flag from the OUTC block.
Field14: crom_reg (9 downto 7): Wait Delay Code (3-bits)

"000" : NOP.
"001" : Use wait delay of 4 clock cycles (for FASC block operation completion).
"010" : Use wait delay of 6 clock cycles (for FMPY block operation completion).
"011" : Use wait delay of 27 clock cycles (for FDIV block operation completion).
"100" : Use wait delay of 78 clock cycles (for SCOS block operation completion).
"101" : Use wait delay of 75 clock cycles (for SEXP block operation completion).
"110" : Use wait delay of 13 clock cycles (for IGEN block operation completion).
"111" : Use wait delay of 2 clock cycles used in an overlapped RESON instruction.

Field15: crom_reg (6): Reset sub-controllers Control (1-bit)

'0' : No Reset.
'1' : Reset all the dedicated controllers of sequential functional units.

Field16: crom_reg (5 downto 3): Bus to Temporary Register (B2TR) (3-bits)

"000" : NOP.
"001" : Load 16-bit integer temporary register (Itemp1) from bus.
"010" : Load 16-bit integer temporary register (Itemp2) from bus.
"011" : Load 32-bit floating-point temporary register (Ftemp1) from bus.
"100" : Load 32-bit floating-point temporary register (Ftemp2) from bus.
"101" : NOP.
"110" : NOP.
"111" : NOP.

Field17: crom_reg (2 downto 0): Temporary Register to Bus (TR2B) (3-bits)

"000" : NOP.
"001" : Put the contents of integer register (Itemp1) onto bus.
"010" : Put the contents of integer register (Itemp2) onto bus.
"011" : Put the contents of floating-point register (Ftemp1) onto bus.
"100" : Put constant 0.5 floating-point register onto bus.
"101" : Put constant 1 integer register onto bus.
"110" : Put constant 0 floating-point register onto bus.
"111" : Put constant 0 integer register onto bus.

6.3.2 Micro-coding from Hardware Flowchart

After writing the hardware flowcharts for all the instructions, the next step is to design the control words (micro-instructions) for all the instructions. Designing of the
control words or micro-instructions from the hardware flowcharts is illustrated below by taking an example of ADDII instruction:

**For State-1:** In the given hardware flowchart, notice the different tasks in the states. According to the sequence of fields in the micro-code format, write the tasks into appropriate fields in encoded form. In our case, first control field is PM (Program Memory) operation; check if there is any PM operation or not. Since there is no PM operation put the value of control field NOP (No OPeration) for PM i.e. “00”. Now check the second field for corresponding operations in the flowchart state and encode the values accordingly and so on. For the hardware flowchart shown below in figure 6.17, various operations are:

1) Increment the PC value by one.
2) Data read from integer memory.
3) Store address from Ptr1_field_of_IR into Ptr1_reg register.
4) Next state is decided by Direct Branch (DB).
5) Select Ptr1_field_of_IR in the instruction register.

Also,
6) present state is Addii-1 and
7) Next state is Addii-2.

![Figure 6.17: State-1 of ADDII Instruction (Showing Various Operations and Annotations)](image-url)
According to the above tasks the respective encodings in the control fields of the corresponding control word are:

- PC increment → “001”
- Select Integer memory → “01”
- Data Read Operation → “00”
- Selection of Ptr1_field_of_IR address pointer → “001”
- DMAU control → “00001”
- Direct branch → “00”
- Next address field → ‘Next Address Value’

Rest of the control fields are NOP and NOP encodings are put in their fields.

For State-2: In state-2 as shown in figure 6.18, the first task shows that the output of the integer memory is loaded into the first input register of the IASC functional block (this is data sourcing) via BUS (this is BUS to integer unit control operation). The second task is data reading from integer memory. Branch type is direct branch.

![Figure 6.18: State-2 of ADDII Instruction](image)

According to the above tasks the respective encodings in the control fields of the corresponding control word are:

- BUS to Functional unit (B2FC) → “0100000” done in IUC
Select Integer memory \Rightarrow \text{“01”}

Data read & source \Rightarrow \text{“11”}

Select Ptr2\_field\_of\_IR \Rightarrow \text{“010”}

DMAU control \Rightarrow \text{“00000”}

Direct branch \Rightarrow \text{“00”}

Next address field \Rightarrow \text{‘Next Address Value’}

Rest of the control fields are NOP and NOP encodings are put in their fields.

**For State-3:** In state-3 as shown in figure 6.19, the output of the integer memory is loaded into second input register of the IASC functional block (this is data sourcing) via BUS (this is BUS to integer unit control operation). Here the type of the operation is also necessary (add or sub). Branch type is direct branch.

![Figure 6.19: State-3 of ADDII Instruction](image)

According to the above task the respective encodings in the control fields of the corresponding control word are:

- BUS to Functional unit (B2FC) \Rightarrow \text{“1000000” done in IUC}
- Select Integer memory \Rightarrow \text{“01”}
- Data source \Rightarrow \text{“10”}
- Direct branch \Rightarrow \text{“00”}
- Next address field \Rightarrow \text{‘Next Address Value’}

Rest of the control fields are NOP and NOP encodings are put in their fields.
For State-4: In state-4 as shown in figure 6.20, the contents of the PC are loaded into the PM for next instruction read. Branch type is direct branch. All the control fields are NOP here except Program Memory Access Control and Next Address Control (NAC).

![Fourth state](image)

**Figure 6.20: State-4 of ADDII instruction**

According to the above task the respective encodings in the control fields of the corresponding control word are:

- Program memory access control ➔ “01”
- Direct branch ➔ “00”
- Next address field ➔ ‘Next Address Value’

Rest of the control fields are NOP and NOP encodings are put in their fields.

For State-5: In state-5 as shown in figure 6.21, the output from the IASC unit is loaded onto the integer memory location at the address given by ptr1_reg. Now this is the end of the execution of the ADDII instruction. Next instruction is loaded after this, so branch type is instruction branch here.

![Fifth state](image)

**Figure 6.21: State-5 of ADDII Instruction**
According to the above tasks the respective encodings in the control fields of the corresponding control word are:

- Functional unit to Bus (F2BC) → “0011”
- Select Integer memory → “01”
- Data write (Read Write Control) → “01”
- Address Selection → “011”
- Next Address Control (IB) → “01”
- Next address field → comes from the Instruction Decoder

Rest of the control fields are NOP and NOP encodings are put in their fields.

In the above example all the steps are explained in detail for developing the control words (micro-instructions) from the hardware flowcharts. Control words for all the instructions of the ASIP have been developed accordingly. The control part is functionally verified by simulating its RT-level VHDL model together with the RT-level model of the datapath (execution unit) for every instruction in the instruction set. Timing of the control part is also verified against the delay specifications provided for it.