AN EXECUTION TRIGGERED COARSE GRAINED RECONFIGURABLE ARCHITECTURE

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF ELECTRICAL AND ELECTRONICS ENGINEERING
AND THE GRADUATE SCHOOL OF ENGINEERING AND SCIENCE
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

By
Oğuzhan Atak
December, 2012

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Abdullah Atalar (Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Erdal Arıkan

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Yalçın Tanık

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Yusuf Ziya İder

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a dissertation for the degree of Doctor of Philosophy.

Prof. Dr. Cevdet Aykanat

Approved for the Graduate School of Engineering and Science:

Prof. Dr. Levent Onural
Director of the Graduate School

ABSTRACT

AN EXECUTION TRIGGERED COARSE GRAINED RECONFIGURABLE ARCHITECTURE

Oğuzhan Atak
PhD in Electrical and Electronics Engineering
Supervisor: Prof. Dr. Abdullah Atalar
December, 2012

In this thesis, we present BilRC (**Bil**kent **R**econfigurable **C**omputer), a new coarse-grained reconfigurable architecture. The distinguishing feature of BilRC is its novel execution-triggering computation model which allows a broad range of applications to be efficiently implemented. In order to map applications onto BilRC, we developed a control data flow graph language, named LRC (a Language for Reconfigurable Computing). The flexibility of the architecture and the computation model are validated by mapping several real world applications. LRC is also used to map applications to a 90nm FPGA, giving exactly the same cycle count performance. It is found that BilRC reduces the configuration size about 33 times. It is synthesized with 90nm technology and typical applications mapped on BilRC run about 2.5 times faster than those on FPGA. It is found that the cycle counts of the applications for a commercial VLIW DSP processor are 1.9 to 15 times higher than that of BilRC. It is also found that BilRC can run the inverse discrete cosine transform algorithm almost 3 times faster than the closest CGRA in terms of cycle count. Although the area required for BilRC processing elements is larger than that of existing CGRAs, this is mainly due to the segmented interconnect architecture of BilRC, which is crucial for supporting a broad range of applications.

Keywords: Coarse-grained Reconfigurable Architectures (CGRA), Discrete Cosine Transform (DCT), Viterbi Decoder, Turbo Decoder, Fast Fourier Transform (FFT), Reconfigurable Computing, Field Programmable Gate Arrays (FPGA) .

ÖZET

YÜRÜTÜME TETİKLEMELİ YENİDEN YAPILANDIRILABİLİR MİMARİ

Oğuzhan Atak
Elektrik Elektronik Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Prof. Dr. Abdullah Atalar
Aralık, 2012

Bu tezde, BilRC olarak adlandırdığımız yeni bir yapılandırılabilir mimari sunuyoruz. BilRC'nin ayırt edici özelliği, geniş bir yelpazedeki uygulamaların etkin bir şekilde gerçeklenmesine imkan sağlayan yürütmeye tetikli hesaplama mimarisidir. Uygulamaları BilRC üzerine yükleyebilmek için LRC (a Language for Reconfigurable Computing) olarak adlandırdığımız bir kontrol data akış diagram dili geliştirildi. Mimarinin ve hesaplama modelinin esnekliği, bir çok uygulamanın BilRC üzerinde gerçeklenmesi ile doğrulandı. LRC dilinde modellenen uygulamalar 90nm teknolojisinde üretilmiş ticari bir FPGA üzerine de yüklendi ve gerçekleme sonuçları karşılaştırıldı. Buna göre, FPGA'yı yapılandırmak için gereken hafıza miktarı BilRC için gereken miktarın ortalama olarak 33 katı olarak bulundu. BilRC 90nm teknolojisinde sentezlendi ve FPGA ile zamanlama karşılaştırması yapıldı. Ortalama olarak BilRC üzerindeki uygulamaların FPGA üzerindeki uygulamalardan 2.5 kat daha hızlı çalıştğı bulundu. BilRC, ticari bir DSP işlemci ile de karşılaştırıldı, DSP üzerinde gerçeklenen uygulamalar için gereken saat çevrim sayısının BilRC için gerekenin 1.9 ile 15 kat arasında olduğu bulundu. BilRC'nin IDCT algorimasını, saat çevrimi açısından, literatürdeki en iyi CGRA'dan 3 kat daha hızlı çalıştırdığı bulundu. BilRC'nin diğer CGRA'lara göre dezavantajı işlem birimlerinin kapladığı alanın diğerlerine göre daha büyük olmasıdır. Bunun temel sebebi BilRC'de kullanılan ara bağlantı hatlarının karmaşıklığıdır.

Anahtar sözcükler: Yeniden Yapılandırılabilir Mimariler, Kesikli Kosinüs Dönüşümü, Viterbi Çözücü, Turbo Çözücü, Hızlı Fourier Dönüşümü, Sahada Programlanabilir Mantık Dizisi.

Acknowledgement

I would like to express my deep gratitude to my supervisor Prof. Abdullah Atalar for his invaluable guidance, support, suggestions and encouragement during the course of this thesis. I am specifically grateful to him that he has installed several Linux servers and Cadence tools by himself to ease my work.

I would also like to thank Prof. Erdal Arıkan for his interesting comments and invaluable suggestions.

I would like to thank my thesis progress and thesis defence jury members Prof. Murat Aşkar, Prof. Yavuz Oruç, Prof. Yalçın Tanık, Prof. Ziya İder and Prof. Cevdet Aykanat for their valuable comments and suggestions.

Finally, I would like to thank my family for their patience and great support.

Dedicated to my parents Duran and Ayşe.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

To comply with the performance requirements of emerging applications and evolving communication standards, various architecture alternatives are available. FPGAs compete with their large number of logic resources. For example, the largest Xilinx Virtex-7 FPGA can provide 6737 GMACS (Giga Multiply and Accumulate per Second) with its 5280 DSP slices[1] and it has 4720 embedded BRAMs each with a 18 Kbits capacity. The main disadvantage of FPGA is the lack of run-time programmability. To maximize the device utilization, FPGA designers partition the available resources among several sub-applications in such a manner that each application works at the chosen clock frequency and complies with the throughput requirement. The design phases of FPGAs and ASICs are quite similar except that ASICs lack post-silicon flexibility. For both FPGAs and ASICs, the function blocks in the application are partitioned to hardware resources spatially.

Unable to exploit the space dimension, DSPs fail to provide the performance requirement of many applications due to the limited parallelism that a sequential architecture can provide. This limitation is not due to the area cost of logic resources, but to lack of a computation model to exploit such a large number of logic resources. Commercial DSP vendors produce their DSPs with several accelerators. For example, Texas Instruments TMS320c6670 DSP has a Turbo

[1] http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

Decoder Coprocessor, FFT and Viterbi decoder accelerators for WCDMA, LTE and WiMAX standards. The disadvantage of such an approach is that the accelerators are designed considering only the applications and standards developed until that time, therefore these accelerators could be useless for emerging applications and evolving standards.

Application-specific instruction-set processors (ASIP) provide high performance with dedicated instructions having very deep pipelines. The basic idea behind the ASIP approach is to shrink the instructions in the loop body into a single or a few instructions so that the number of cycles spent for the loop kernel is reduced. For example, the FFT processors presented in [1, 2, 3, 4] have special instructions for the FFT kernel. ASIPs are designed in general for a specific algorithm or algorithms having similar computation kernel. For example, an ASIP [5] with a 15-pipeline stage is presented for various Turbo and convolutional code standards. A Multi-ASIP [6] architecture is presented for exploiting different parallelism levels in the Turbo decoding algorithm. The basic limitation of the ASIP approach is its weak programmability, which makes it inflexible for emerging standards. For instance, aforementioned ASIPs do not support Turbo codes with more than 8-states [6] and 16-states [5]. In order to make ASIPs flexible after fabrication, reconfigurable ASIPs (rASIP) have been proposed [7] having programmable function generators similar to that of FPGAs.

Coarse-grained reconfigurable architectures (CGRA) have been proposed to provide a better performance/flexibility balance than the alternatives discussed above. Hartenstein [8] compared several CGRAs according to their interconnection networks, data path granularities and application mapping methodologies. In a recent survey paper, De Sutter *et al.* [9] classified several CGRAs according to computation models while discussing the relative advantages and disadvantages. Compton *et al.* [10] discussed reconfigurable architectures containing heterogeneous computation elements such as CPU and FPGA, and compared several fine- and coarse-grained architectures with partial and dynamic configuration capability. According to the terminologies used in the literature [8, 9, 10], reconfigurable architectures (RA), including FPGAs, can be classified according to the configuration in three distinct models as single-time configurable, statically

reconfigurable and dynamically reconfigurable. Statically reconfigurable RAs are configured at loop boundaries, whereas dynamic RAs can be configured at almost each clock cycle. The basic disadvantage of statically reconfigurable RAs is that if the loop to be mapped is larger than the array size, it may be impossible to map. However, the degree of parallelism inside the loop body can be decreased to fit the application to CGRA. This is the same approach that designers use for mapping applications to an FPGA. In dynamically reconfigurable RAs, the power consumption can be high due to fetching and decoding of the configuration at every clock cycle. However, techniques have been proposed [11, 12] to reduce power consumption due to dynamic configuration. The interconnect topology of RAs can be either one-dimensional (1D) such as PipeRench [13, 14, 15] and RAPID [16, 17] or two-dimensional (2D) such as ADRES [18, 19, 20, 21, 22], MorphoSys [23], MORA [24, 25], REMARC [26], GARP [27, 28], KressArray[29, 30], RAW [31], MATRIX [32], COLT [33], PACT XPP [34, 35, 36] and conventional FPGAs.

RAs can have a point to point (p2p) interconnect structure as in ADRES, MORA, MorphoSys and PipeRench or a segmented interconnect structure as in KressArray, RAPID and conventional FPGAs. p2p interconnect has the advantage of deterministic timing performance. The clock frequency of the RA does not depend on the application mapped while the fanout of the Processing Elements (PE) is limited. If an operation has more sinks than the interconnects allow, one of the PEs is used to delay the data for one clock cycle. Limited p2p interconnect may increase the initiation interval [20] and cause performance degradation. For the segmented interconnect method, the output of a PE can be routed to any PE, while the timing performance depends on the application mapped. For FPGAs, the timing closure is similar to that of ASICs and is quite tedious, whereas for a segmented-interconnect CGRA timing closure is rather simple due to coarser granularity.

The execution control mechanism of RAs can be either of a statically scheduled type such as MorphoSys and ADRES, where the control flow is converted to data flow code during compilation, or a dynamically scheduled type such as KressArray, which uses tokens for execution control.

In this thesis, we present BilRC[2], a statically reconfigurable CGRA with a 2D segmented interconnect architecture utilizing dynamic scheduling with execution triggering. KressArray is the most similar architecture with some basic differences: First, KressArray uses a data-driven execution control mechanism together with a centralized sequencer, whereas BilRC with no centralized controller, the execution control is distributed. Second, KressArray uses a dynamic global bus for both primary input/output and temporary data transfer in between PEs, and local static interconnect for PE communication, BilRC uses a segmented static interconnect for all communication requirements. Third, KressArray does not have any multiplier and memory unit in the array architecture which limits the applications that can be mapped on. BilRC, like FPGAs, have memory and multiplier PEs so that almost all applications can be implemented. Our contributions can be summarized as follows:

- An execution triggered computation model is presented and the suitability of the model is validated with several real world applications. For this model, a language for reconfigurable computing, LRC, is developed.
- A new CGRA employing segmented interconnect architecture with three types of PEs and its configuration architecture is designed in 90nm CMOS technology. The CGRA is verified up to the layout level.
- Full tool flow including a compiler for LRC, a cycle accurate SystemC simulator and a placement & routing tool for mapping applications to BilRC are developed.
- CGRAs are known to reduce configuration size, however there is no work on configuration size comparison of CGRAs and FPGAs. The applications modeled in LRC are converted to HDL with our LRC-HDL converter and then mapped onto an FPGA and to BilRC on a-cycle-by-cycle equivalent basis. Then, a comparison of precise configuration size and timing is done.
- It is known that CGRAs can provide better timing performance as compared to FPGAs. However, there is no work on comparing the timing

[2]BilRC: Bilkent Reconfigurable Computer

performance of the two. Thanks to LRC and LRC-HDL generator, the critical path for several applications are found for both FPGA and BilRC for a timing performance comparison.

- The segmented interconnect structure is rather mature for FPGAs, however the required number of tracks (ports) for CGRAs has not been explored yet. We used state of the art placement and routing heuristics to minimize the number of ports required to implement several applications with challenging communication requirements.

The rest of the thesis is organized as follows: In Chapter 2, the architecture of PEs and the configuration mechanism are presented. Chapter 3 discusses the execution triggered computation model. In Chapter 4, the tools developed for application mapping to BilRC and FPGA are explained. In Chapter 5, mapping of a number of applications to BilRC is presented. The physical implementation results, cycle count performance, the critical path performance and a configuration size comparison are given in Chapter 6. The thesis is concluded in Chapter 7.

Chapter 2

BilRC Architecture

BilRC has three types of PEs: *Arithmetic logic unit* (ALU), *memory* (MEM) and *multiplier* (MUL). Similar to the some commercial FPGA architectures such as Stratix[1] and Virtex[2], PEs of the same type are placed in the same column as shown in Fig 2.1. The architecture repeats itself every nine columns and the number of rows can be increased without changing the distribution of PEs. This PE distribution is obtained by considering several benchmark algorithms from signal and image processing and telecommunication applications. The PEs' distribution can be adjusted for better utilization for the targeted applications. For example, the Turbo decoder algorithm does not require any multiplier, but needs a large amount of memory. On the other hand, filtering applications require many multipliers, but not much memory. For the same reason, commercial FPGAs have different families for logic-intensive and signal processing-intensive applications.

2.1 Interconnect Architecture

PEs in BilRC are connected to four neighboring PEs [2] by communication channels. Channels at the periphery of the chip can be used for communicating with

[1] http://www.altera.com
[2] http://www.xilinx.com

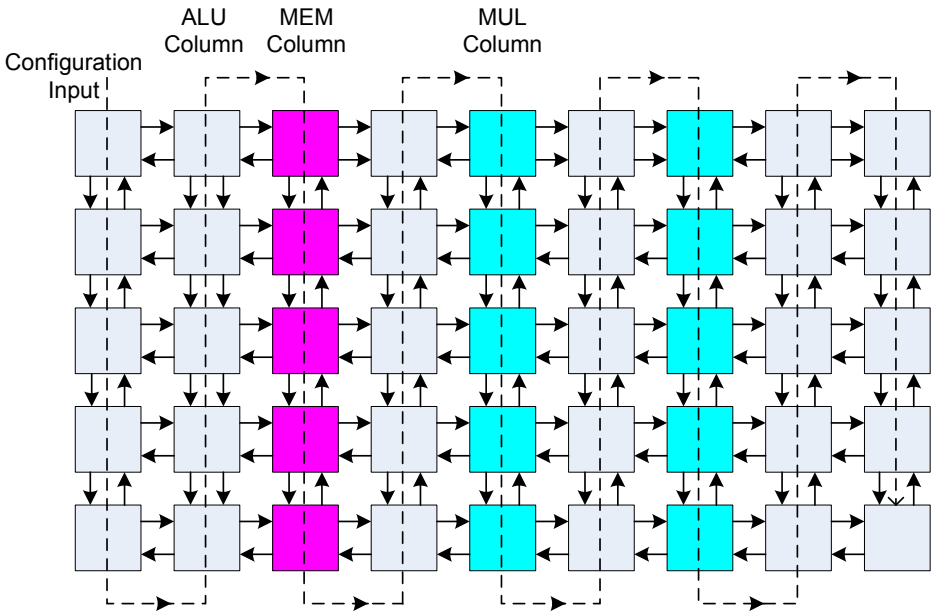

Figure 2.1: Columnwise allocation of PEs in BilRC

the external world.

If the number of ports in a communication channel is N_p, the total number of ports a PE has is $4N_p$. The interconnect architecture is the same for all PE types. Fig. 2.2 illustrates the signal routing inside a PE for $N_p = 3$. There are three inputs and three outputs on each side. The output signals are connected to corresponding input ports of the neighbor PEs. The input and output signals are all 17 bits wide. 16 bits are used as data bits and the remaining *Execute Enable* (EE) bit is used as the control signal.

PEs contain *processing cores* (PC) located in the middle. *Port route boxes* (PRB) at the sides are used for signal routing. PCs of ALUs and MULs have two outputs and the PC of MEM has only one output. Each PC output is a 17 bit signal. The second output of a PC is utilized for various purposes, such as the execution control for loop instructions, the carry output of additions, the most significant part of multiplication, the maximum value of index calculation and the conditional execution control. PC outputs are routed to all PRBs. Therefore, any PRB can be used to route PC output in the desired direction. All input signals

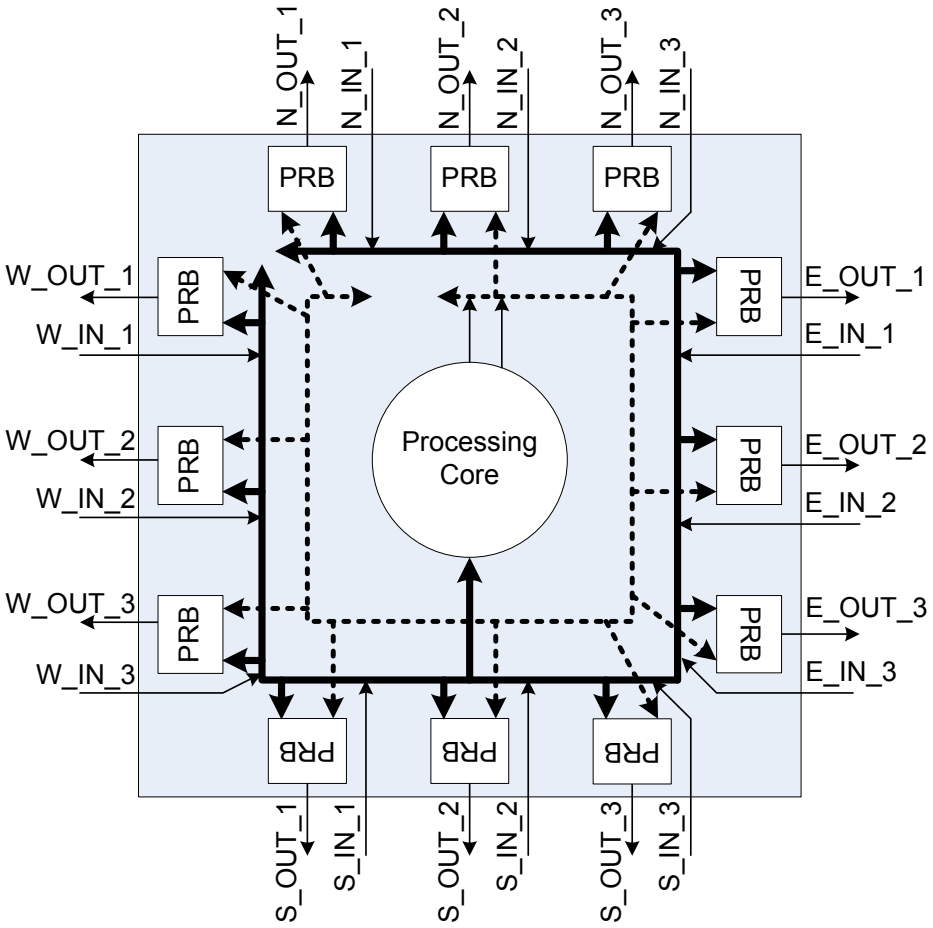

Figure 2.2: Input/Output Signal Connections

are routed to all PRBs and to the PC as shown in Fig. 2.2. The PC selects its operands from the input signals by using internal multiplexers. Fig. 2.3 shows the internal structure of PRB. The Route multiplexer is used to select signals coming from all input directions and from the PC. The pipeline multiplexer is used to optionally delay the output of the route multiplexer for one clock cycle. The idea of using multiplexers for signal routing has already been used in [37]. BilRC is configured statically, hence both the interconnects and the instructions programmed in PCs remain unchanged during the run.

Fig. 2.4 shows an example mapping. PE_1 is the source and PE_4 is the destination while PE_2 and PE_3 are used for signal routing. It must be noted that the pipelining elements are not used. Inside PC_1 an instruction is executed and

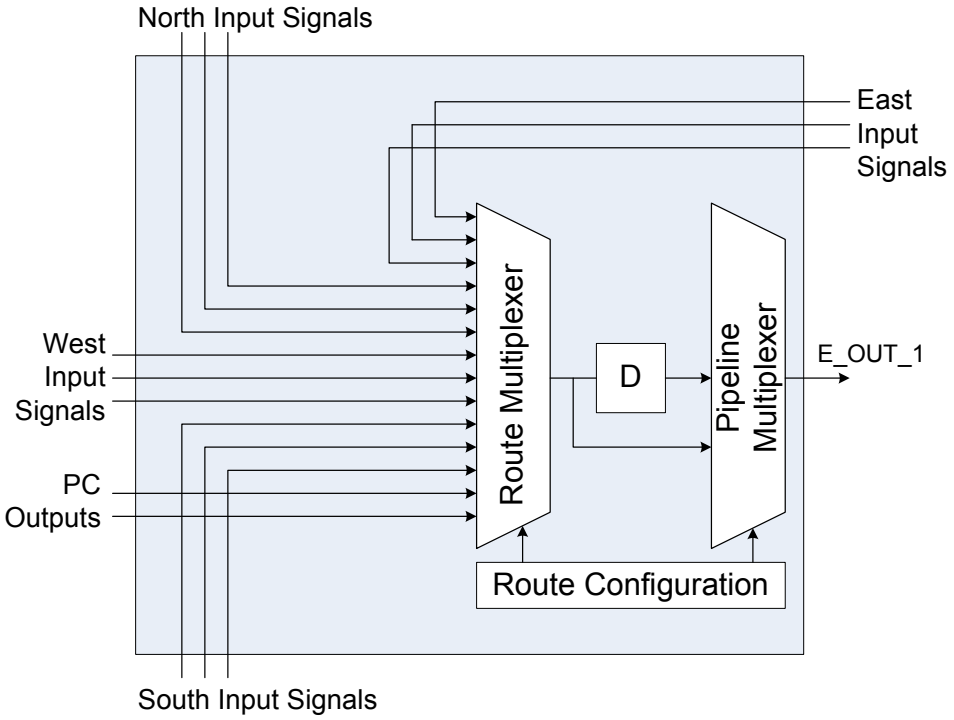

Figure 2.3: Schematic Diagram of PRB

the result is registered. The critical path starts from the output of the source PC. Then, the signal is routed through PE_2 and PE_3. T_{HOP} is the time delay to traverse a PE (without using the pipelining element in PRB). Finally, the signal at PE_4 goes through the adder and reaches the output register in PC with a time delay of T_{PE}. The total delay, T_{CRIT}, between the register in PE_1 and the register in PE_4 is given as

$$T_{CRIT} = nT_{HOP} + T_{PE} \tag{2.1}$$

where n=2 is the number of hops, T_{HOP} is the time delay to traverse one PE and T_{PE} is the time delay within a PE.

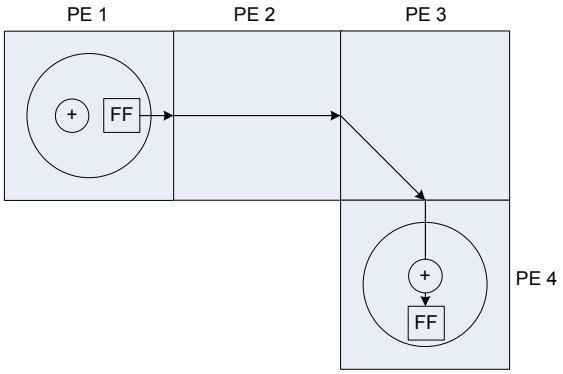

Figure 2.4: An example of routing between two PEs.

2.2 Processing Core Architectures

2.2.1 MEM

Fig. 2.5 shows the architecture of the processing core of MEM. PC has a data bus which is formed from all input data signals and an execute enable bus which is formed from all input EE signals. SRAM block in PC is a 1024×16 dual port RAM (10 address bits, 16 data bits). `op1_adr` set by the configuration register determines which one of the 12 inputs is the read address. Similarly, `op2_adr` chooses one of the inputs as the write address. The most significant six bits are compared with MemID stored in the configuration register. If they are equal, then read and/or write operations are performed. `opr3_addr` selects the data to be written from one of the input ports. One of the input ports of SRAM is used only for writing and the other one is used only for reading. The read address and read enable signals are selected by `op1_adr` from the data bus and the execute enable bus, respectively. The least significant 10 bits of data are used as the read address for SRAM and the most significant 6 bits are used as the Memory ID (MemID). MemID is used to form larger memory arrays by using multiple MEMs. If MemID in the data bus is equal to MemID in the configuration register, the data at location addressed by the read address signal is read and the output execute enable signal, (`PC_OUT_1_EE`), is enabled. If MemIDs are not equal, the output signal is disabled. The write address and write enable signals are selected by `op2_adr` in a similar way.

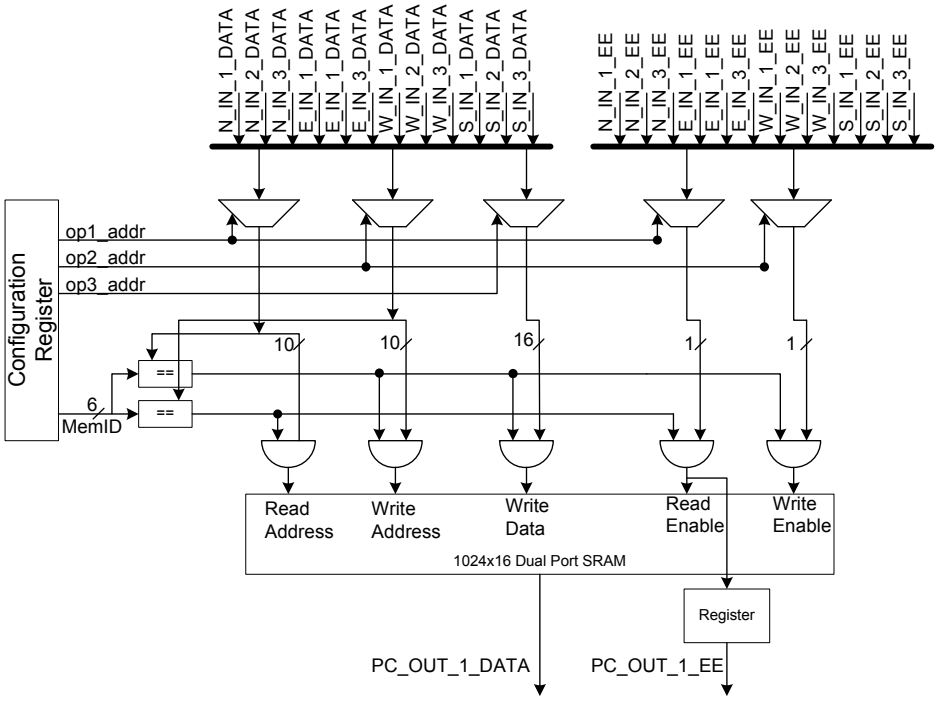

Figure 2.5: Processing Core Schematic of MEM

2.2.2 ALU

Fig. 2.6 shows the architecture of ALU. Similar to MEM, ALU has two buses for input data and execute enable signals. The instruction to be executed in ALU is programmed during configuration and the ALU executes the same instruction during application run. The operands to the instructions are selected from the data bus by using the multiplexers M3, M4, M5, M6. ALU has an 8×16 register file for storing constant data operands. For example, an ALU with the instruction, `ADD(A,100)` reads the variable `A` from an input port and the constant `100` is stored in the register file during configuration. The output of the register file is connected to the data bus so that the instruction can select its operand from the register file. The execution of the instruction is controlled from the execute enable bus. The configuration register has a field to select the input execute enable signal from the execute enable bus. PC executes the instruction when the selected signal is enabled.

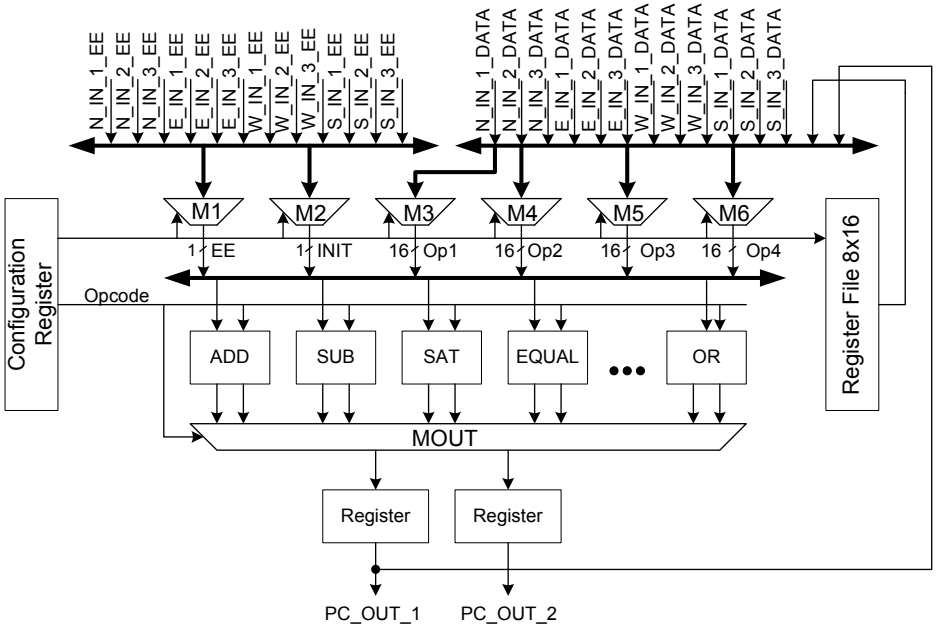

Figure 2.6: Processing Core Schematic of ALU

2.2.3 MUL

The processing core of MUL is similar to that of ALU. The difference is the instructions supported in the two types of PEs. Multiplication and shift instructions are performed in this PE. The `MUL` instruction performs the multiplication operation on two operands. The operands can be from the inputs (variable operands) or from the register file (constant operands). The result of the multiplication is a 32-bit number that appears on two output ports. The most significant part of the multiplication is put on the second output, and the least significant part is put on the first output. Alternatively, the result of the multiplication can be shifted to the right in order to fit the result to a single output port by using the `MUL_SHR` (multiply and shift to the right) instruction. This instruction executes in two clock cycles: the multiplication is performed in the first clock cycle and the shifting is performed in the second clock cycle. The rest of the instructions for all PEs are executed in a single clock cycle. The shift operation is performed by a barrel shifter. The remaining instructions supported in MUL are the instructions requiring a barrel shifter.

Table 2.1: Configuration data structure

Conf. Item	number of words	Meaning
PID	1	Processing Element ID
N	1	Number of words in the configuration packet
Configuration Register (CR)	3	PC configuration register
Route Configuration Register (RCR)	5	It is used to configure multiplexers in the PRBs
Output Initialization Register	1	loads the register for output initialization
Register File or Memory Content Configuration	variable	The register file of ALU or MEM or the SRAM of the MEM is initialized

2.3 Configuration Architecture

PEs are configured by configuration packets which are composed of 16-bit configuration words. Table 2.1 lists the data structure of the configuration packet. Each PE has a 16-bit-wide configuration input and a configuration output. These signals are connected in a chain structure as shown in Fig. 2.1. The first word of the configuration packet is the processing element ID (PID). It is used to address the configuration packet to a specific PE. A PE receiving the configuration packet uses it if the PID matches its own ID, otherwise it is forwarded to the next PE in the chain. The second word in the packet is the length of the configuration packet, this word is useful for register and memory initializations to indicate the size of the configuration packet. The configuration register (CR) is used to configure PC. The fields of the CR are illustrated in Table 2.2 for ALU. The configuration register of MEM does not require the fields `opr4_adr`, `EE_adr`, `Init_Addr`, `Init_type` and `Init_Enable`, and the configuration register of MUL does not contain the `opr4_addr` field, since none of the instructions require four operands. CR is 48 bits long for all PC types; the unused bit positions are reserved for future use. It must be noted that the bit width of the configuration register and the route configuration register depends on N_p. The number of words

Table 2.2: ALU Configuration Register

Conf. Field	number of bits	Meaning
opr1_addr	5	Operand 1 Address
opr2_addr	5	Operand 2 Address
opr3_addr	5	Operand 3 Address
opr4_addr	5	Operand 4 Address
EE_addr	5	Execute Enable Input Address
Init_addr	4	Initialization Input Address
op_code	8	Selects the instruction to be executed
Init_Enable	1	Determines whether the PC has an initialization or not
Init_Type	1	Determines the type of the initialization

for the fields given in the table is for N_p=4.

Chapter 3

Execution-Triggered Computation Model

Writing an application in a high-level language, such as `C` and then mapping it on the CGRA fabric is the ultimate goal for all CGRA devices. To get the best performance from the CGRA fabric, a middle-level language (assembly-like language) that has enough control on PEs and provides abstractions is necessary. The designers thus do not deal with unnecessary details, such as the location of the instructions in the 2D architecture and the configuration of route multiplexers for signal routing. Although there are compilers for some CGRAs which directly map applications written in a high-level language such as C to the CGRA [38, 34, 39, 28], the designers still need to understand the architecture of the CGRA in order to fine tune applications written in C-code for the best performance [9].

The architecture of BilRC is suitable for direct mapping of control data flow graphs (CDFG). A CDFG is the representation of an application in which operations are scheduled to the nodes (PEs) and dependencies are defined. We developed a *Language for Reconfigurable Computing* (LRC) for the efficient representation of CDFGs. In this thesis, it is assumed that the CDFG is available, generating CDFGs from a high level language is out of the scope of this work. Existing tools such as IMPACT [21] can be used to generate a CDFG in the form of

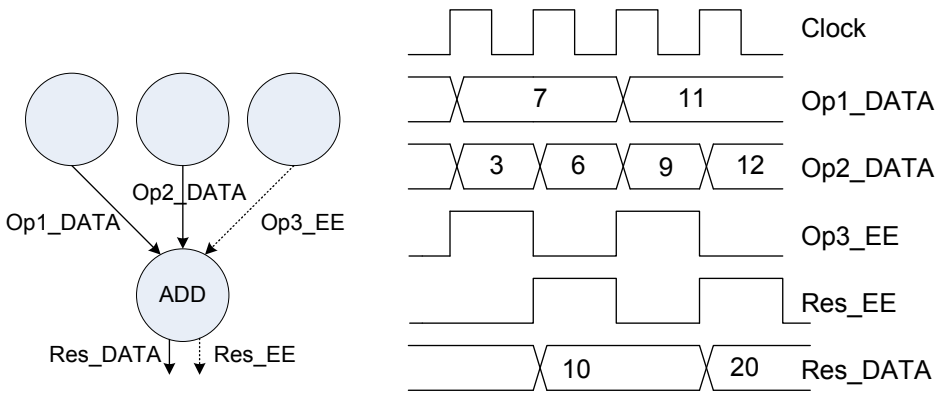

Figure 3.1: Example CDFG and Timing Diagram

an intermediate representation called LCode. IMPACT reads a sequential code, draws a data flow graph and generates a representation defining the instructions that are executed in parallel. Such a representation can then be converted to an LRC code.

3.1 Properties of LRC

3.1.1 LRC is a spatial language

Unlike sequential (imperative) languages, the order of instructions in LRC is not important. LRC instructions have execution control inputs that trigger the execution. LRC can be considered as a graph drawing language in which the instructions represent the nodes and the data and control operands (dependencies) represent the connections between the nodes.

3.1.2 LRC is a single assignment language

LRC is a functional language similar to Single-Assignment-C language [40, 41]. During mapping to the PEs, each LRC instruction is assigned to a single PE. Therefore, the output of the PEs must be uniquely named. A variable can be

assigned to multiple values indirectly in LRC by using the self-multiplexer instruction, `SMUX`. Examples for `SMUX` are provided in Chapter 3.3.2 and Chapter 5.7.

3.1.3 LRC is cycle accurate

In LRC, the number of clock cycles spent for the execution of an instruction is deterministic. Each instruction in LRC, except `MUL_SHR`, is executed in a single clock cycle. Therefore, even before mapping to the architecture, cycle-accurate simulations are possible to obtain timing diagrams of the application.

3.1.4 LRC has an execution-triggering mechanism

LRC instructions have explicit control signal(s), which trigger the execution of instruction assigned to the node. Instructions that are triggered from the same control signal execute concurrently, hence parallelism is explicit in LRC, i.e., the application designer can control the degree of parallelism.

3.2 Advantages of Execution Triggered Computation Model

The execution-triggered computation model can be compared to the data flow computation model [42]. The basic similarity is that both models build a data flow graph such that nodes are instructions and the arcs between the nodes are operands. The basic difference is that the data flow computation model uses tagged tokens to trigger execution; a node executes when all its operands (inputs) have a token and the tags match. Basically, tokens are used to synchronize operands and tags are used to synchronize different loop iterations. In LRC an instruction is executed when its execute enable signal is active. Application of the data flow computation model to CGRAs has the following problems: first, tagged tokens require a large number of bits; this in turn increases the interconnect area.

For example, the Manchester Machine [42] uses 54 bits for tagged tokens. Second, a queue is required to store tagged tokens which increases the area of PE. Third, a matching circuit is required for comparing tags, both increasing PE area and decreasing performance. For example, an instruction with three operands requires two pairwise tag comparisons to be made. Execution-triggered computation uses a single bit as execute enable; hence it is both area efficient and fast.

The execution-triggered computation model can be compared to the computation models of existing CGRAs. MorphoSys [23] uses a RISC processor for the control-intensive part of the application. The reconfigurable cell array is intended for the data-parallel and regular parts of the application. There is no memory unit in the array; instead, a frame buffer is used to provide data to the array. The RISC processor performs loop initiation and context broadcast to the array. Each reconfigurable cell runs the broadcast instructions sequentially. This model has many disadvantages. First, an application cannot be always partitioned into control-intensive and data-intensive parts, and even if it is partitioned, the intercommunication between the array and RISC creates a performance bottleneck. Second, the lack of memory units in the array limits the applications that can be run on the array. Third, the loop initiation is controlled by the RISC processor, hence the array can be used only for innermost loops.

ADRES[21] uses a similar computation model with some enhancements, the RISC processor is replaced with a VLIW processor. ADRES is a template CGRA. Different memory hierarchies can be constructed by using the ADRES core. For example, two levels of data caches can be attached to ADRES [22], or a multi-ported scratch pad memory can be attached [43, 44]. There is no array of data memories in the ADRES core. The VLIW processor is responsible for loop initiation and the control-intensive part of the application. Lack of parallel data memory units in the ADRES core limits the performance of the applications mapped on ADRES. For example, 8-state Turbo decoder algorithm requires at least 13 memory units for efficient implementation, as explained in Chapter 5.7. In a recent work on ADRES [43], a 4-ported scratchpad memory was attached to the ADRES core for applications requiring parallel memory accesses. BilRC targets more parallelism levels than does ADRES. In our recent work [2], we

have shown that it is possible to map an LDPC decoder that requires 24 parallel memory accesses in a single clock cycle. In ADRES, the loops are initiated from the VLIW processor. Hence, only a single loop can run at a time. ADRES has a mature tool suite, which can map applications written in C-language direcly to the architecture. Obviously, this is a major advantage. The VLIW processor in the ADRES can also be used for the parts of the applications which require low parallelism.

MORA [25] is intended for multimedia processing. The reconfigurable cells are DSP-style sequential execution processors, which have internal 256-byte data memory for partial results and a small instruction memory for dynamic configuration of the cells. The reconfigurable cells communicate with an asynchronous handshaking mechanism. MORA assembly language and the underlying reconfigurable cells are optimized for streaming multimedia applications. The computation model is unable to adapt to complex signal processing and telecommunications applications.

RAPID [17] is a one-dimensional array of computation resources, which are connected by a configurable segmented interconnect. RAPID is programmed with RAPID-C programming language. During compilation the application is partitioned into static and dynamic configurations. The dynamic control signals are used to schedule operations to the computation resources. A sequencer is used to provide dynamic control signals to the array. The centralized sequencer approach to dynamically change the functionality requires a large number of control signals, and for some applications the required number of signals would not be manageable. Therefore, RAPID is applicable to highly regular algorithms with repetitive parts.

LRC is more efficient than the computation model of existing CGRAs from a number of perspectives:

1. LRC has flexible and efficient loop instructions. Therefore, no external RISC or VLIW processor is required for loop initiation. Arbitrary number of loops can be run in parallel. The applications targeted for LRC are not

limited to the innermost loops. For example, the IDCT algorithm has two loops one for horizontal and one for vertical processing, these loops can be pipelined so that after the first loop finishes the two loops run in parallel. Another example is that the turbo decoding algorithm has two loops one for processing the received symbols in the normal order and one for processing the received symbols in the interleaved order. Moreover these loops has two inner loops one for processing data in the forward direction and one for processing data in the reverse order. Such complex loop topologies can be easily modeled in LRC.

2. LRC has memory instructions to flexibly model the memory requirements of the applications. For example, the Turbo decoding algorithm requires 13 memory units. The access mechanism to the memories is efficiently modeled. The extrinsic information memory in the Turbo decoder is accessed by four loop indices. LRC has also flexible instructions to build larger-sized memory units. ADRES, MorphoSys and MORA have no such memory models in the array.

3. The execution control of LRC is distributed. Hence, there is no need for an external centralized controller to generate control signals, as is required in RAPID. The instruction set in LRC is flexible enough to generate complex addressing schemes, and no external address generators are required. While LRC is not biased to streaming applications, they can be modeled easily. It must be noted that LRC is not biased to any specific application, i.e., there are no application specific instructions.

3.3 Modeling Applications in LRC

In a CDFG, every node represents a computation, and connections represent the operands. An example CDFG and timing diagram is shown in Fig. 3.1. The node `ADD` performs an addition operation on its two operands `Op1_Data` and `Op2_Data` when its third operand, `Op3_EE`, is activated. Here, `Op1` and `Op2` are data operands and `Op3` is a control operand. Below is the corresponding LRC line.

```
[Res,0]=ADD(Op1,Op2)<-[Op3]
```

In LRC, the outputs are represented between the brackets on the left of the equal sign. A node can have two outputs; for this example only the first output, `Res`, is utilized. A "0" in place of an output means that it is unused. `Res` is a 17-bit signal that is composed of 16-bit data, `Res_Data`, and a 1-bit execute enable signal, `Res_EE`. The name of the function is provided after the equal sign. The operands of the function are given between the parentheses. The control signal that triggers the execution is provided between the brackets on the right of the "`<-`" characters. As can be seen from the timing diagram, the instruction is executed when its execute enable input is active. The execution of an instruction takes one clock cycle; therefore, the `Res_EE` signal is active one clock cycle after `Op3_EE`.

3.3.1 Loop Instructions

Signal processing and telecommunication algorithms contain several loops which are in nested, sequential or parallel topology. For example, FFT algorithm has a nested loop in which the outer loop counts the stages in the algorithm and the inner loop counts the butterflies within a stage. The loops are responsible for a great portion of the execution time. Therefore, efficient handling of loops is critical for the performance of most applications. LRC has flexible and efficient loop instructions. By using multiple LRC loop instructions, nested, sequential and parallel loop topologies can be modeled. A typical `FOR` loop in LRC is given as follows:

```
[i,i_Exit]=FOR_SMALLER(StartVal,EndVal,Incr)<-[LoopStart,Next]
```

This `FOR` loop is similar to that in C-language:

```
for(i=StartVal;i<EndVal;i=i+Incr)
   {loop body}
```

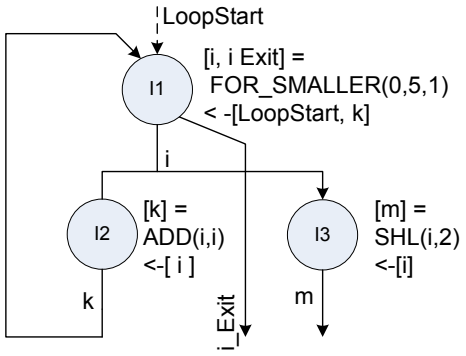

Figure 3.2: CDFG and LRC example for FOR_SMALLER

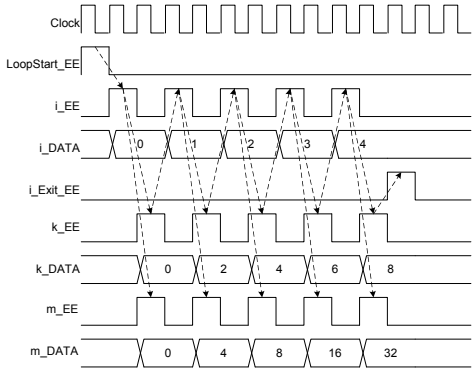

Figure 3.3: Timing Diagram of FOR_SMALLER

The `FOR_SMALLER` instruction works as follows:

- When the LoopStart signal is enabled for one clock cycle, the data portion of the output, `i_DATA`, is loaded with `StartVal_DATA`, and the control part of the output `i_EE` is enabled in the next clock cycle.
- When the `Next` signal is enabled for one clock cycle, `i_DATA` is loaded with `i_DATA+Incr_DATA` and `i_EE` is enabled if `i_DATA+Incr_DATA` is smaller than `EndVal`; otherwise, `i_Exit_EE` is enabled.

The parameters `StartVal`, `EndVal` and `Incr` can be variables or constants.

Fig. 3.2 shows an example CDFG having three nodes. The LRC syntax of the instructions assigned to the nodes is shown at the right of the nodes. All operands of `FOR_SMALLER` are constant in this example. When mapped to PEs,

constant operands are initialized to the register file during configuration. ADD and SHL (SHift Left) instructions are triggered from i_EE. Hence, their outputs k and m are activated at the same clock cycles as illustrated in Fig. 3.3. The Next input of the FOR_SMALLER instruction is connected to the k_EE output of the ADD instruction. Therefore, FOR_SMALLER generates an i value for every two clock cycles. When i exceeds the boundary, FOR_SMALLER activates the i_Exit signal. The triggering of instructions is illustrated in Fig. 3.3 with dotted lines. SFOR_SMALLER is a self-triggering FOR instruction given as

```
[i,i_Exit]=SFOR_SMALLER(StartVal,EndVal,Incr,IID)<-[LoopStart]
```

The SFOR_SMALLER instruction does not require a Next input; but instead it requires a fourth constant operand, IID (Inter Iteration Dependency). SFOR_SMALLER waits for the IDD cycles to generate the next loop index after generating the current loop index. This instruction triggers itself and can generate an index for every clock cycle when IID is 0. LRC has support for loops whose index variables are descending; these instructions are FOR_BIGGER and SFOR_BIGGER. The aforementioned for loop instructions can be used as a *while* loop by setting the Incr operand to 0. By doing so, it always generates an index value. This is equivalent to an infinite while loop. The exit from this while loop can be coded externally by conditionally activating the Next input.

3.3.2 Modeling Memory in LRC

In LRC, every MEM instruction corresponds to a 1024-entry, 16-bit, 2-ported memory. One port is used for writing data to memory and the other port is used for reading data from the memory. The syntax for MEM instruction is given below:

```
[Out]=MEM(MemID,ReadAddr,InitFileName,WriteAddr,WriteIN)
```

The MEM instruction takes five operands. MemID is used to create larger memories as discussed earlier. ReadAddr is the read address port of the memory. This

signal is composed of ReadAddr_Data and ReadAddr_EE signals. The 10 least significant bits of ReadAddr_Data are connected to the read address port of the memory. When ReadAddr_EE is active, the data in the memory location addressed by ReadAddr_Data is put on Out_DATA in the following clock cycle and Out_EE is activated. The InitFileName parameter is used for initializing the memory. The write operation is similar to reading. When WriteAddr_EE is active, the data in WriteIN_Data is written to the memory location addressed by WriteAddr_Data. Program 1shows the code for forming a 2048-entry memory: The first memory

```
1: [Out1]=MEM(0,ReadAddr,File0,WriteAddr,WriteData)
2: [Out2]=MEM(1,ReadAddr,File1,WriteAddr,WriteData)
3: [Out]=SMUX(Out1,Out2)
```

Program 1: Building a 2048-Entry Memory in LRC

has MemID=0. This memory responds to both read and write addresses if they are between 0 and 1023; similarly, the second memory responds only to the addresses between 1024 and 2047. Therefore, the signals Out1_EE and Out2_EE cannot both be active in the same clock cycle. The SMUX instruction in the third line multiplexes the operand with the active EE signal. Due to the SMUX instruction, one clock cycle is lost. The SMUX instruction can take four operands. Therefore, up to 4^n memories can be merged with n clock cycles of latency.

3.3.3 Conditional Execution Instructions

Conditional executions are inevitable in almost all kinds of algorithms. Although some signal processing kernels such as FIR filtering do not require conditional executions, an architecture without conditional executions can not be considered complete. LRC has novel conditional execution control instructions. Below is a conditional assignment statement in C language:

```
if(A>B) {result=C;} else{result=D;}
```

Its corresponding LRC code is given as

```
[c_result,result]=BIGGER(A,B,C,D)<-[Opr]
```

BIGGER executes only if its execute enable input, Opr_EE, is active. result is assigned to operand C if A is bigger than B; otherwise it is assigned to D. c_result is activated only if A is bigger than B. Since c_result is activated only if the condition is satisfied, the execution control can be passed to a group of instructions that is connected to this variable. The example C code below contains not only assignments, but also instructions in the if and else bodies.

```
if(A>B) {result=C+1;} else {result=D-1;}
```

This C-code can be converted to an LRC code by using three LRC instructions as shown in Program 2. The first line evaluates C+1, the second line evaluates D-1

```
1: [Cp1,0]=ADD(C,1)<-[C]
2: [Dm1,0]=SUB(D,1)<-[D]
3: [0,result]=BIGGER(A,B,Cp1,Dm1)<-[Opr]
```

Program 2: Use of Comparison Instruction in LRC

and in the third line, result is conditionally assigned to Cp1 or Dm1 depending on the comparison A>B. Conditional instructions supported in BilRC are as follows: SMALLER, SMALLER_EQ (smaller or equal), BIGGER, BIGGER_EQ (bigger or equal), EQUAL and NOT_EQUAL. By using these instructions, all conditional codes can be efficiently implemented in LRC. ADRES [19] uses a similar predicated execution technique. In LRC two branches are merged by using a single instruction. In a predicated execution, a comparison is made first to determine the predicate, and then the predicate is used in the instruction. In LRC, the results of two or more instructions cannot be assigned to the same variable, since these instructions are the nodes in the CDFG. Therefore, the comparison instructions in LRC are used to merge two branches of instructions. Similar merge blocks are used in data flow machines [42] as well.

The conditional assignment instructions in LRC is summarized in Table-3.1.

Table 3.1: Conditional Assignment Instructions in LRC

C Language Syntax	LRC Instruction
>	BIGGER
>=	BIGGER_EQ
<	SMALLER
<=	SMALLER_EQ
==	EQUAL
! =	NOT_EQUAL

3.3.4 Initialization Before Loops

```
1: min=32767;
2: for(i=0;i<255;i++){
3:    A=mem[i];
4:    if(A<min) {min=A;}
5: }
```

Program 3: Minimum value of an array in C

In the C-code in Program 3, the variable `min` is assigned twice, before the loop and inside the loop. Such initializations before loops are frequently encountered in applications with recurrent dependencies. Multiple assignment to a variable is forbidden in LRC as discussed in Chapter 3.1.2. An initialization technique has been devised for LRC instructions, which removes the need for an additional `SMUX` instruction.

The corresponding LRC code is given below: `MIN` finds the minimum of its

```
1: [i,i_Exit]=SFOR_SMALLER(0,256,1,0)<-[LoopStart]
2:    [A,0]=MEM(0,i,filerand.txt,WriteAddr,WriteData)
3:    [min(32767),0]=MIN(min,0,A,0)<-[A,LoopStart]
```

Program 4: Minimum value of an array in LRC

first and third operands[1]. The execute enable input of the `MIN` instruction is `A_EE`. The second control signal between the brackets to the right of the "$<-$" characters, `LoopStart`, is used as the initialization enable. When this signal is active, the `Data` part of the first output is initialized. The parentheses after the output signal `min` represent the initialization value.

3.3.5 Delay Elements in LRC

CDFG representation of algorithms requires many delay elements. These delay elements are similar to the pipeline registers of pipelined processors. A value calculated in a pipeline stage is propagated through the pipeline registers so that further pipeline stages use the corresponding data.

```
1: for(i=0;i<256;i++){
2:     A=mem[i];
3:     B=abs(A);
4:     C=B>>1;
5:     if(C>2047){R=2047;}
6:     else{R=C;}
7:     res_mem[i]=R;
8: }
```

Program 5: Pipelinining

In the C-code in Program 5, the data at location `i` is read from a memory `A`, its absolute value is calculated at `B`, shifted to the right by 1 at `C` and finally saturated and saved to the memory at location `i`. The corresponding LRC code is given in Program 6.

Although the LRC instructions are written in Program 6 in the same order as in the C-code in Program 5, this is not necessary. The order of instructions in LRC is not important. The `IID` operand of the `SFOR_SMALLER` instruction is set to 0. Therefore, an index value, `i`, is generated from 0 to 255 at every clock cycle, i.e., software pipelining [45] is used. After six clock cycles, all the instructions

[1]The second and fourth operands of `MIN` are used for the index of minimum calculation.

```
1: [i,i_Exit]=SFOR_SMALLER(0,256,1,0)<-[LoopStart]
2: [A,0]=MEM(0,i,filerand.txt,0,0)
3: [B,0]=ABS(A)<-[A]
4: [C,0]=SHR(B,0,1)<-[B]
5: [0,R]=BIGGER(C,2047,2047,C)<-[C]
6: [mem2,0]=MEM(0,0,0,i(4),R)
```

Program 6: Pipelining in LRC

are active at each clock cycle until the loop boundary is reached. Since the instructions are pipelined, the `MEM` instruction above cannot use `i` as the write address, but its four-clock-cycle delayed version. The number of pipeline delays is coded in LRC by providing it between the parentheses following the variable. It must be noted that the number of pipeline delays are constant and it must be determined at design time. A variable for a pipeline delay is not allowed, since these delay elements are part of the interconnection network which are fixed after the configuration. The requirement to specify delay value explicitly in LRC for pipelined designs makes code development a bit difficult. However, the difficulty is comparable to that of designing with HDL or assembly languages.

3.3.6 Utilization of the Second Output

In LRC, some of the instructions have two outputs. The second output is used for a number of purposes. Although the basic Processing Core architecture is 16-bit, i.e., the operands of the instructions are 16-bits, it is possible to create larger size arithmetic. One purpose of the second output is as the carry output of an addition:

```
1: [R_lsb,carry]    = ADD(A_lsb,B_lsb)<-[A_lsb]
2: [R_msb,0]        = ADDC(A_msb(1),B_msb(1),carry)<-[A_lsb(1)]
```

Program 7: Utilization of the second output as the carry signal

In the code in Program 7, `A_lsb` and `A_msb` represent the LSB and MSB

parts of an 32-bit signal. The instruction `ADDC` has an additional third operand `carry`. The first two operands are delayed one clock cycle to match them with the `carry` signal. It must be noted that only the least significant bit of the signal `carry` is utilized. However, routing a dedicated carry line in the interconnection network would be more problematic since this line is only utilized by the addition instruction. In BilRC, the second output of the PC is used for different purposes for different instructions, and it is routed in the interconnection network only if it is required.

The second output can also be utilized for finding the index of maximum of an array. In Program 8, a tree is formed by using the `MAX` instructions.

```
1:[max_01,ind_01]=MAX(A0,0,A1,1)<-[A0]
2:[max_23,ind_23]=MAX(A2,2,A3,3)<-[A2]
3:[max_45,ind_45]=MAX(A4,4,A5,5)<-[A4]
4:[max_67,ind_67]=MAX(A6,6,A7,7)<-[A6]
5:[max_03,ind_03]=MAX(max_01,index_01,max_23,ind_23)<-[max_01]
6:[max_47,ind_47]=MAX(max_45,index_45,max_67,ind_67)<-[max_45]
7:[max_07,ind_07]=MAX(max_03,index_03,max_47,ind_47)<-[max_03]
```

Program 8: Utilization of second output for finding index of maximum

Chapter 4

Tools and Simulation Environment

Fig. 4.1 illustrates the simulation and development environment. The four key components are:

4.1 LRC Compiler

Takes the code written in LRC and generates a pipelined netlist. Every instruction in LRC corresponds a node in CDFG which is assigned to a PC in BilRC and every connection between two nodes is a net. The net has the following information: input connection, output connection, the number of pipeline stages between the input and the output.

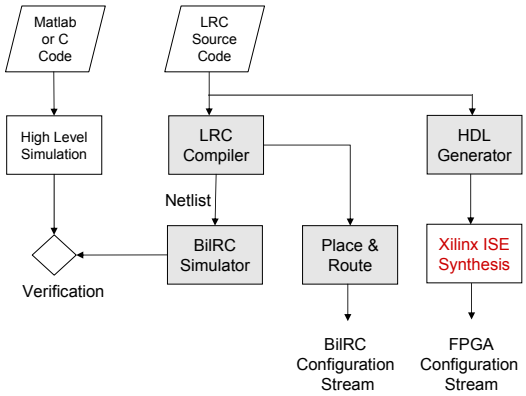

Figure 4.1: Simulation and Implementation Environment

4.2 BilRC Simulator

Performs cycle-accurate simulation of LRC code. BilRC simulator was written in SystemC[1]. The pipelined netlist is used as the input to BilRC simulator. PCs are interconnected according to the nets. If a net in the netlist file has delay elements, then these delay elements are inserted between PCs. The results of a simulation can be observed in three ways: from the SystemC console window, the Value Change Dump (VCD) file or the BilRC log files. Every PC output has been registered to SystemC's built-in function `sc_trace`; thus by using a VCD viewer all PC output signals can be observed in a timing diagram.

4.3 Placement & Routing Tool

This tool maps the nodes of CDFGs into a two-dimensional architecture, and finds a path for every net. Since the interconnection architecture of BilRC is similar to that of FPGAs, similar techniques can be used for placement and routing. However, unlike that of FPGAs, the interconnection network of BilRC is pipelined. This is the basic difference between FPGA and BilRC interconnection networks. BilRC place & route tool finds the location of the delay elements during the placement phase. The placement algorithm uses the simulated annealing

[1]http://www.systemc.org/home/

technique with a cooling schedule adopted from [46]. The total number of delay elements that can be mapped to a node is $4N_p$. For every output of a PC, a pipelined interconnect is formed. When placing the delay elements, contiguous delay elements are not assigned to the same node. Such movements in the simulated annealing algorithm are made forbidden. A counter is assigned for every node, which counts the number of delay elements assigned to the node. The counter values are used as a cost in the algorithm. Therefore, delay elements are forced to spread around the nodes. The placement algorithm uses the shortest path tree algorithm for interconnect cost calculation. The algorithm used for routing is similar to that of the negotiation based router [47]. Fig. 5.2 shows the result of placement and routing of the maxval algorithm explained in Chapter 5.1.

4.4 HDL generator

Converts LRC code to HDL code. Since LRC is a language to model CDFGs, it is easy to generate the HDL code from it. For each instruction in LRC, there is a pre-designed VHDL code. The HDL generator connects the instructions according to the connections in the LRC code. The unused inputs and outputs of instructions are optimized during HDL generation. The quality of the generated HDL code is very close to that of manual coded HDL. The generated HDL code can then be used as an input to other synthesis tools, such as the Xilinx ISE. The generated HDL code was used to map applications to an FPGA in order to compare the results with LRC code mapped to BilRC.

Chapter 5

Example Applications for BilRC

In order to validate the flexibility and efficiency of the proposed computation model, several standard algorithms selected from Texas Instruments benchmarks [48] are mapped to BilRC. We also mapped Viterbi and Turbo decoder channel decoding algorithms and multirate and multichannel FIR filters. For all cases, it is assumed that the input data are initialized into the memories and the outputs are directly provided to the device outputs.

5.1 Maximum Value of an Array (maxval)

The maximum value of an array can be computed in LRC in different ways depending on how the array stored in memories. The input array of size 128 is stored in 8 sub-arrays with a size of 16 each. The algorithm first finds the maximum values of the 8 sub-arrays by sequentially processing each data read from the memories, and then the maximum value from among these 8 values are computed. Fig. 5.1 illustrates the CDFG of the algorithm.

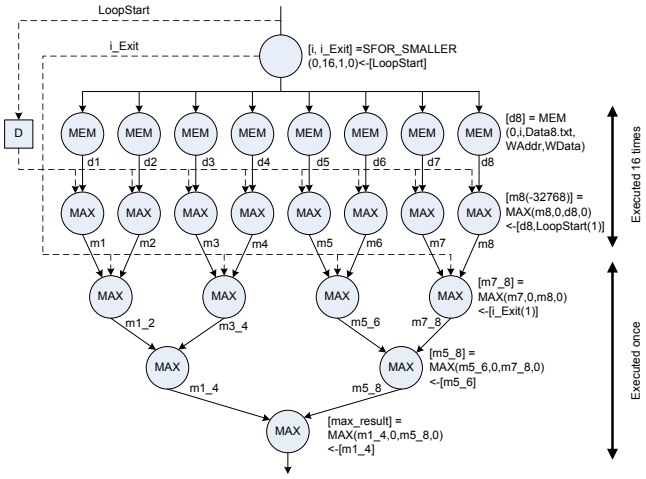

Figure 5.1: LRC Code and CDFG of Maximum Value of an Array

```
1: [LoopStart]=DELAY(PI)<-[PI]
2: [i, i_Exit] = SFOR_SMALLER( 0,16,1,0)<-[LoopStart]
3:    [d.1]       = MEM(0,i,Data1.txt,0,0)<-[]
4:    [d.2]       = MEM(0,i,Data2.txt,0,0)<-[]
5:    ...
6:    [d.8]    = MEM(0,i,Data8.txt,0,0)<-[]
7:    [m1(-32768)] = MAX(m1,0,d1,0)<-[d1,LoopStart(1)]
8:    [m2(-32768)] = MAX(m2,0,d2,0)<-[d2,LoopStart(1)]
9:    ...
10:   [m8(-32768)] = MAX(m8,0,d8,0)<-[d8,LoopStart(1)]

11:     [m1_2] = MAX(m1,0,m2,0)<-[i_Exit(1)]
12:     [m3_4] = MAX(m3,0,m4,0)<-[i_Exit(1)]
13:     [m5_6] = MAX(m5,0,m6,0)<-[i_Exit(1)]
14:     [m7_8] = MAX(m7,0,m8,0)<-[i_Exit(1)]
15:     [m1_4] = MAX(m1_2,0,m3_4,0)<-[m1_2]
16:     [m5_8] = MAX(m5_6,0,m7_8,0)<-[m5_6]
17:     [max_result] = MAX(m1_4,0,m5_8,0)<-[m1_4]
```

Program 9: Maximum Value of an Array

The signal, LoopStart, triggers the SFOR_SMALLER instruction. The loop generates an index value for every clock cycle, starting from 0 and ending at 15. i is used as an index to read data from 8 memories in parallel. Then, 8 MAX instructions find the maximum values corresponding to each sub-array. The instruction corresponding to the eighth sub-array is shown below:

```
[m8(-32768)]=MAX(m8,0,d8,0)<-[d8,LoopStart(1)]
```

Here, the variable m8 is both output and input. At every clock cycle, m8 is compared to d8 and the larger one is assigned to m8. The LoopStart(1) signal (1 in parentheses indicates one clock cycle delay) is used to initialize m8 to -32768. It should be noted that if an instruction's output is also input to itself, the output variable is connected to the input bus inside the processing core. This is shown in Fig. 2.6, where PC_OUT_1 is connected to the input data bus. During compilation, LRC compiler finds the instructions whose output is also input, and then the PE is configured accordingly.

When the FOR loop reaches the boundary, i_Exit_EE is activated for one clock cycle, one-cycle-delayed version of i_Exit_EE is used to trigger the execution of four MAX instructions. The dotted lines in the figure represent the control signals and the solid lines represent signals with both control and data parts. The instructions in the MAX-tree are executed only once. The depth of the memory blocks in BilRC is 1024, whereas the maxval algorithm uses only 16 entries. This under-utilization of memory can be avoided by using register files instead of memories. ALU PEs have 8-entry register files, two ALU PEs can be used to build a 16 entry register file.

5.2 Dot Product of two Vectors

This algorithm can be computed on BilRC in different ways depending on how the input vectors are stored. It will be assumed that the vectors a and b are stored in 8 memories. Thus, there are 8 sub arrays. In the LRC code given in

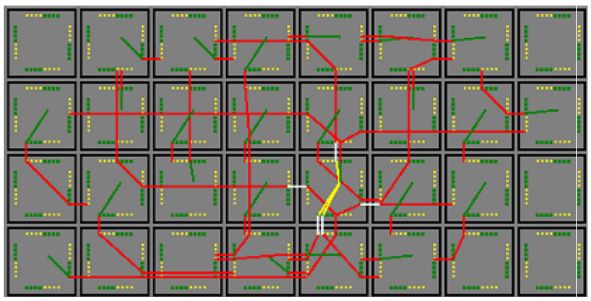

Figure 5.2: Maxval algorithm placement and routing on BilRC

Appendix C.3, first 8 dot products of the sub arrays are computed. Then, these partial dot products are summed up. This example shows the utilization of the second output of the `MUL_SHIFT` instruction. The least and most significant parts of the multiplications are accumulated for each loop iteration. The `carry` output resulting from the adder is used as an input for the MSB part. Since the LSB addition takes one clock cycle, the MSB part of the multiplication is delayed one clock cycle to balance the two inputs.

5.3 Finite Impulse Response Filters

Digital filters can be implemented by using a tap delay line, multipliers and an adder tree. A 16-tap FIR filter can be described in LRC as given in the Program 10. In this example, it is assumed that both the filter input data which is stored in a memory and the filter coefficients are represented as 12-bit signed values. The write address and data ports of the memory are not used in this example. In a real implementation, these ports are used or the filter input data can be read from a primary input. `SFOR_SMALLER` instruction in the first line generates an index at every clock cycle. This index value is used as the address of the memory in the second instruction. 16 `MUL_SHIFT` instructions multiply the coefficients with the filter input data and shift the result to the right by 11. The second multiplier, `mul1` uses one clock cycle delayed version of `data`, and the 16^{th} multiplier uses 15 clock cycle delayed version of `data`. The tap delay line is implicitly defined in LRC by using the delayed versions of the input `data`. The

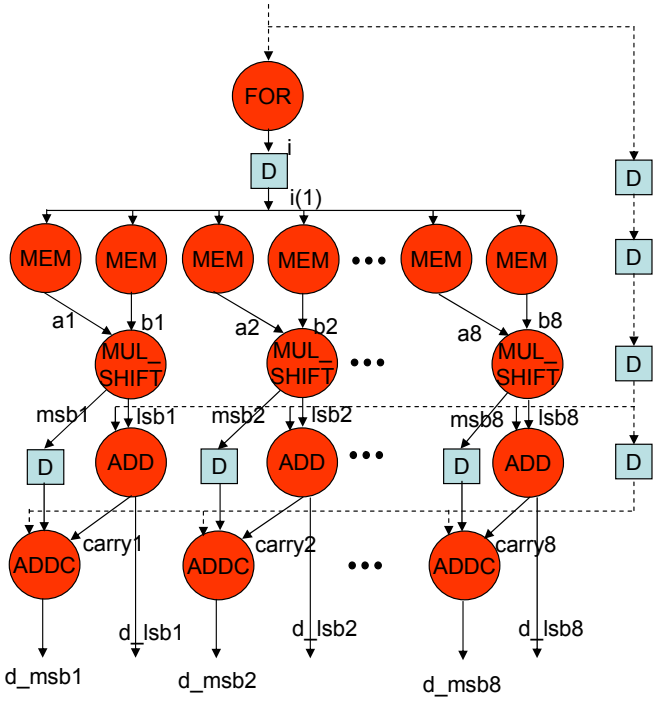

Figure 5.3: Part of the CDFG of dot product algorithm

results of the multiplication are used as input to an adder tree.

A multi-rate filter can be designed with LRC in a similar way. In order to design a multi-rate filter with rate 2, the `MUL_SHIFT` instructions in Program 10 can be changed as given in Program 11. In this code, the second multiplier uses `data(2)`, which is two clock cycle delayed version of `data` and third multiplier uses `data(4)` and so on. As compared to the single rate FIR, the number of delay elements in the algorithm is doubled. Since BilRC has plenty of delay elements, this does not create a problem.

The multi-rate filter described above can be used as a multichannel filter by multiplexing channel data at the filter input and demultiplexing the data at the filter output. The multiplexing at the filter input is shown in Program 12. In this code, `SFOR_SMALLER` instruction generates an index for every two clock cycles, since its 4^{th} operand, `IID`, is set to 1. The memory for the `data_ch1` uses `i` as the address and `data_ch2` uses `i(1)`, one clock cycle delayed version of `i`. `data_ch1` and `data_ch2` are active for one clock cycle for every two clock cycle. The output

```
[i, i_Exit] = SFOR_SMALLER( 0,1024,1,0)<-[LoopStart]\\
    [data]    = MEM(0,i,data.txt,0,0)<-[]
    #multiplication by filter coefficients
    [mul0]    = MUL_SHIFT(data    ,-23 ,11)<-[data]
    [mul1]    = MUL_SHIFT(data(1) ,-39 ,11)<-[data(1)]
    ...
    [mul15 ]  = MUL_SHIFT(data(15),-23 ,11)<-[data(15)]
    #adder tree
    [add1_0]  = ADD(mul0 ,mul1)<-[mul0]
    ...
    [add1_7]  = ADD(mul14,mul15)<-[mul14]
    [add2_0]  = ADD(add1_0 ,add1_1)<-[add1_0]
    ...
    [add2_3]  = ADD(add1_6 ,add1_7)<-[add1_6]
    [add3_0]  = ADD(add2_0 ,add2_1)<-[add2_0]
    [add3_1]  = ADD(add2_2 ,add2_3)<-[add2_2]
    [filter_out]  = ADD(add3_0 ,add3_1)<-[add3_0]
```

Program 10: FIR Filter

```
[mul0 ]  = MUL_SHIFT(data    ,-23 ,11)<-[data]
[mul1 ]  = MUL_SHIFT(data(2) ,-39 ,11)<-[data(2)]
[mul2 ]  = MUL_SHIFT(data(4) ,-39 ,11)<-[data(2)]
...
[mul15 ] = MUL_SHIFT(data(30),-23 ,11)<-[data(30)]
```

Program 11: Part of the Multi-Rate FIR Filter

of SMUX is active for every clock cycle and contains data from the first channel for one clock cycle and from the second channel in the following clock cycle.

```
[i,i_Exit]  = SFOR_SMALLER( 0,1024,1,1)<-[LoopStart]
[data_ch1]  = MEM(0,i,data.txt,0,0)<-[]
[data_ch2]  = MEM(0,i(1),data.txt,0,0)<-[]
[data]      = SMUX(data_ch1,data_ch2)<-[]
```

Program 12: Part of the Multi-Channel FIR Filter

5.4 2D-IDCT Algorithm

2D-DCT and its inverse, 2D-IDCT algorithms are widely used in image processing for compression and decompression respectively. In this work, we consider the implementation of (8x8) 2D-IDCT algorithm with LRC. We used a fixed point model of the Program [48]. The algorithm is composed of three parts: horizontal pass, transposition and vertical pass. In the horizontal pass, the rows of the 8×8 matrix are read and the 8-point 1D IDCT of the row is computed. Since there are 8 rows in the matrix, this operation is repeated 8 times. The transposition phase of the algorithm transposes the resulting matrix obtained from the horizontal pass. In the final phase, the matrix is read again row-wise and the 1D IDCT of each row is computed. The challenging part of the algorithm is the transposition phase.

Fig. 5.4 illustrates the CDFG and LRC of the algorithm. This algorithm computes 2D-IDCT of 100 frames, where a frame is composed of 64 words. The code assumes that the input data is stored in 8 arrays. While the input arrays are being filled, the IDCT computation can run concurrently. Hence, the time to get data to the memory can be hidden. The two SFOR_SMALLER instructions at the beginning of the code are used for frame counting and horizontal line counting, respectively. The SHR_OR instruction computes the address, which is used to read data from the eight memory locations. MUX (multiplex) instructions in the code are used for transposition. The MUX instruction has five operands: the first

```
[frame_cnt ]  = SFOR_SMALLER( 0,100,1,8)<-[LoopStart]
    [hor_cnt ]    = SFOR_SMALLER( 0,8,1,0)<-[frame_cnt]
    [hor_addr]    = SHL_OR(frame_cnt,3,hor_cnt)<-[hor_cnt]
#Read a Row
    [data_1]      = MEM(0,hor_addr,data_1.txt,0,0)<-[]
    ...
    [data_8]      = MEM(0,hor_addr,data_8.txt,0,0)<-[]
#Horizontal IDCT computations
    ...
    [reg1_wd_m1]=MUX(sel10,f0,f1(1),f2(2),f3(3))   <-[sel10]
    [reg1_wd_m2]=MUX(sel10,f4(4),f5(5),f6(6),f7(7))<-[sel10]
    [reg1_wd]   =MUX(sel3,reg1_wd_m1,reg1_wd_m2,0,0)<-[sel3]
    ...
    [reg8_wd_m1]=MUX(sel10(7),f0,f1(1),f2(2),f3(3))
                 <-[sel10(7)]
    [reg8_wd_m2]=MUX(sel10(7),f4(4),f5(5),f6(6),f7(7))
                 <-[sel10(7)]
    [reg8_wd]   =MUX(sel3(7),reg8_wd_m1,reg8_wd_m2,0,0)
                 <-[sel3(7)]
    ...
#Vertical IDCT computations
    ...
```

Program 13: IDCT

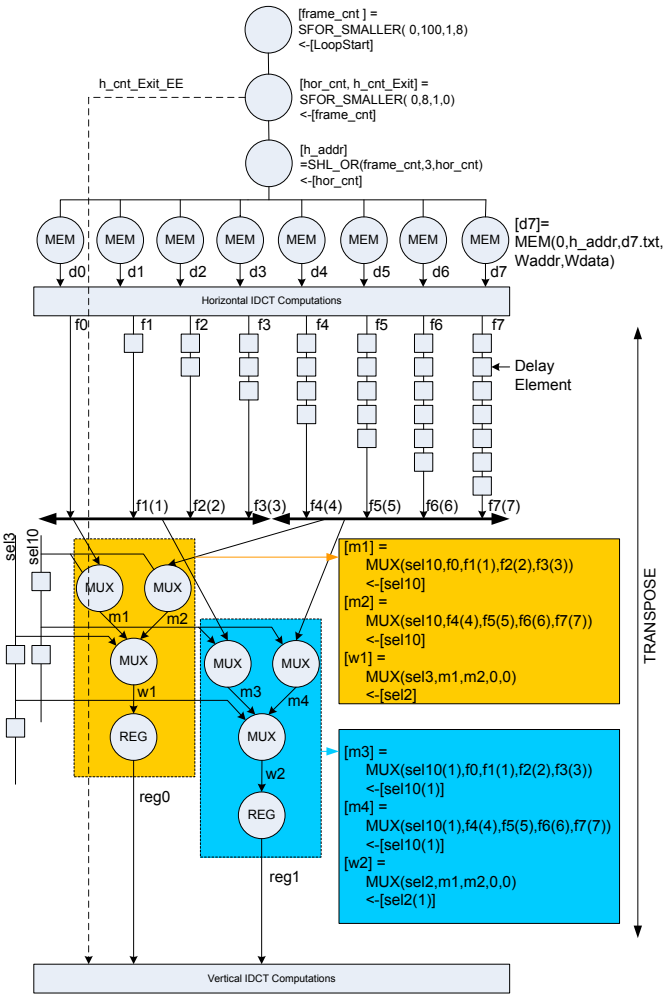

Figure 5.4: LRC code and CDFG of 2D-IDCT Algorithm

operand is used as the selection input, and the remaining four operands are to be multiplexed. In order to multiplex eight operands, three multiplexers are used. The variables `[f0,f1,...,f7]` are the results of the horizontal IDCT. These variables are used as the input operands of the multiplexers. `f0` is connected to the input of the multiplexer directly, whereas `f1` is delayed one clock cycle; hence `f1(1)` and `f2` is delayed two cycles. The horizontal results are queued in a pipeline for the first register, `reg0`. For the second register, `reg1`, `sel10` and `sel3`, which are selection operands of the multiplexers, are delayed, so that the second horizontal results are queued. The transposition operation is performed by using 24 `MUX` instructions and 31 delay elements.

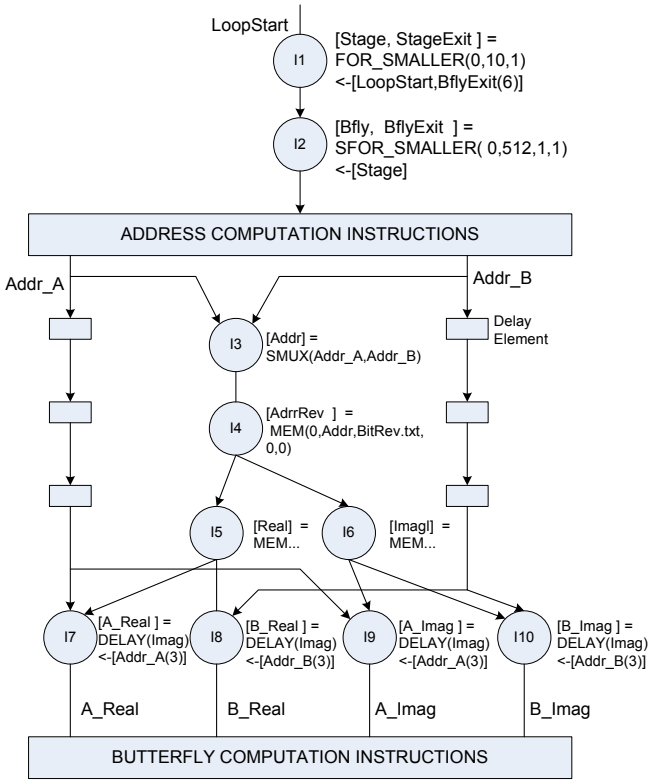

Figure 5.5: LRC code and CDFG of FFT

The `IID` parameter of the `SFOR_SMALLER` instruction for horizontal line counting is set to 0. Therefore, an index is generated every clock cycle, and computation of eight horizontal IDCTs takes 8 clock cycles. The computation of the vertical IDCTs takes 8 clock cycles as well. The computations of horizontal and vertical IDCTs are pipelined. Thus, a 2D-IDCT is computed in 9 clock cycles on the average (1 clock cycle is lost in loop instructions). The computation of 100 frames takes only 930 clock cycles.

5.5 FFT Algorithm

FFT algorithm is widely used in signal processing and telecommunication applications. We have designed 1024 point radix-2 DIT (Decimation In Time) FFT algorithm in LRC. This algorithm is computed in two loops: the outer loop counts

the stages and the inner loop counts the butterflies in the stages. For 1024 point FFT there are $\log_2(1024) = 10$ stages and $1024/2 = 512$ butterflies. A butterfly has two inputs from the data memory and a coefficient from the coefficient memory. The LRC code and the CDFG is illustrated in 5.5. The code takes 39 LRC instructions. The loop for stage counting uses `BflyExit(6)` signal which is the 6 clock delayed version of the exit signal of the loop for butterfly counting. The inner loop generates an index for every two clock cycle, since the data memory is sequentially accessed by the two inputs of a butterfly. The signals `bflyadr_a` and `bflyadr_b` are the addresses of the butterfly. These signals are active once for every two clock cycles. They are active non overlapping clock cycles. The signal `DataAddr` which is the output of `SMUX` instruction is active for every clock cycle. DIT FFT algorithm accesses the memory in bit reverse order, a memory, which is initialized with bit reversed addresses is used for this purpose. `AdrrRev` is the bit reverse address of the butterfly and is used as the read address for the data memory. The input data to the FFT is represented in two memory locations for real and imaginary parts. The real and imaginary data read from the memory are demultiplexed by using `DELAY` instructions. `oprA_Real` and `oprB_Real` are the real parts of the two inputs of a butterfly. The effectiveness of the LRC for accessing a shared resource, the memory in the current example, must be noted. In VHDL such access mechanisms are generally coded with complex state machines. The butterfly computation is composed of multiplication, addition and subtraction instructions. The results of the butterfly computation are saved to the memory locations where the inputs of the butterfly is read.

5.6 Viterbi Decoder

In communications, Viterbi decoding [49] is used to decode convolutionally encoded information. The algorithm is computed in two phases. In the first phase, probabilities of Markov states are computed recursively in the forward direction, i.e., in the same direction the information is encoded. At each step, the survivor paths are stored. In the second phase, the stored paths are traversed backwards and at each step a symbol is decoded, i.e., the most probable value of a symbol

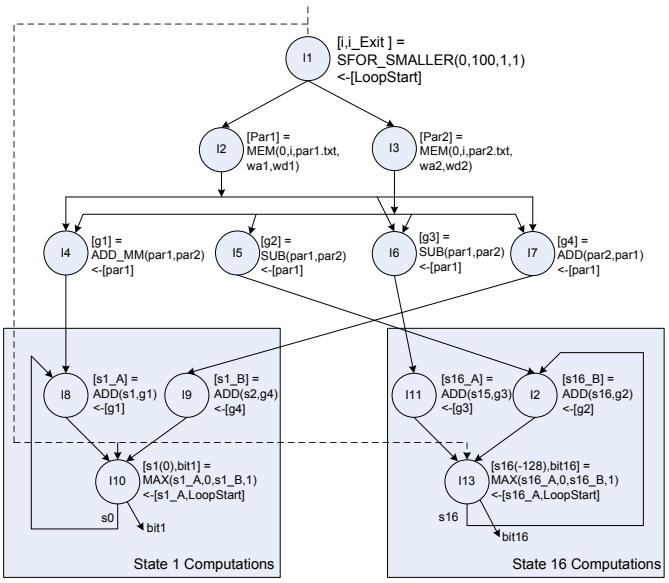

Figure 5.6: LRC code and CDFG of Viterbi Decoder

is decided.

The Viterbi algorithm given in this section is for a convolutional code with 16 states, i.e., the constraint length is 5 and the code rate is 1/2.

In Fig. 5.6, the forward computation phase of the algorithm is illustrated. In this code, the frame size is 100 and the rate of the encoding is `1/2`. The number of Markov states is 16. `IID` parameter of the `SFOR_SMALLER` (`I1`) is set to 1. Therefore, an index value is generated for every two clock cycles. In the second and third instructions, the two parity likelihoods are read from two memory locations. From these likelihood values four path metrics are calculated, `G0`, `G1`, `G2`, `G3`. These four path metrics corresponds to four combinations of the parity bits, i.e., 00, 01, 10 and 11. Then, these path metrics are used to calculate the two accumulating state path metrics for each state. The initial value of `S0` is set to 0 and remaining states are set to -128. Here, -128 corresponds to $-\infty$ meaning that the probability of being in this state is almost impossible. `LoopStart` signal is used to initialize the state signals. `MAX` finds the maximum value of its first and third operands and puts the result on the first output. If the greater operand is the first one, it puts the second operand on the second output, otherwise the fourth operand is put on the second output.

```
1:[i, i_Exit ]     = SFOR_SMALLER( 0,100,1,1)<-[LoopStart]
2:[Sys]            = MEM(0,i,r_sys.txt,0,0)<-[]
3:[Par]            = MEM(0,i,r_par.txt,0,0)<-[]
4:[G0]             = ADD\_MM(Par,Sys)<-[Sys]
5:[G1]             = SUB   (Par,Sys)<-[Sys]
6:[G2]             = SUB   (Sys,Par)<-[Sys]
7:[G3]             = ADD   (Sys,Par)<-[Sys]
8:[s0_metric_0]    = ADD(S0,G0)<-[G0]
9:[s0_metric_1]    = ADD(S1,G1)<-[G1]
10:[s1_metric_0]   = ADD(S2,G0)<-[G0]
11:[s1_metric_1]   = ADD(S3,G1)<-[G1]
12:[s2_metric_0]   = ADD(S0,G3)<-[G3]
13:[s2_metric_1]   = ADD(S1,G2)<-[G2]
14:[s3_metric_0]   = ADD(S2,G3)<-[G3]
15:[s3_metric_1]   = ADD(S3,G2)<-[G2]
16:[S0(0),bit0]    = MAX(s0_metric_0,0,s0_metric_1,1)
                           <-[s0_metric_0,LoopStart]
17:[S1(-128),bit1] = MAX(s1_metric_0,0,s1_metric_1,1)
                           <-[s1_metric_0,LoopStart]
18:[S2(-128),bit2] = MAX(s2_metric_0,0,s2_metric_1,1)
                           <-[s2_metric_0,LoopStart]
19:[S3(-128),bit3] = MAX(s3_metric_0,0,s3_metric_1,1)
                           <-[s3_metric_0,LoopStart]
```

Program 14: Forward Recursion of the Viterbi Algorithm

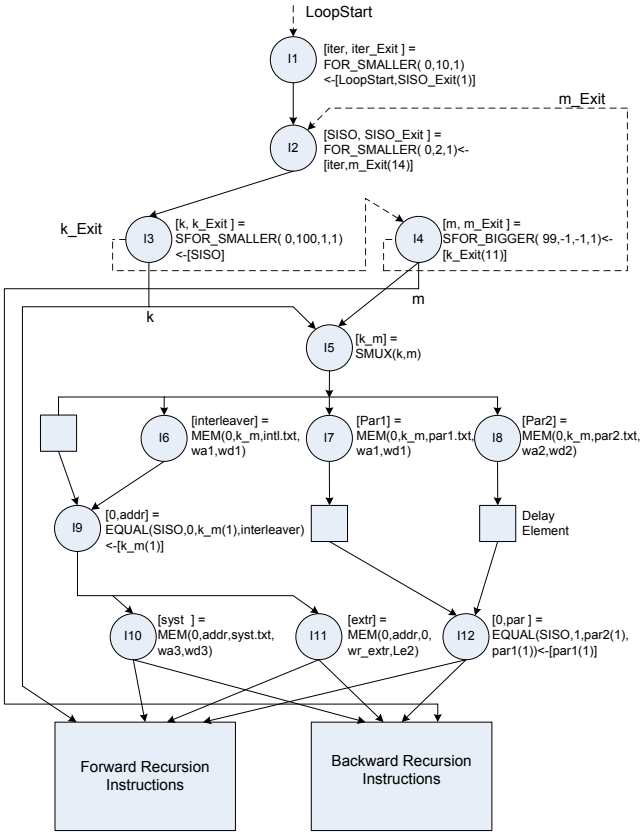

Figure 5.7: LRC code and CDFG of UMTS Turbo Decoder

The backward computation of the algorithm is simpler than the forward phase. Starting from the most probable state, the algorithm decodes a bit for every 3 clock cycles. At each step the most probable state is decided from the surviving path and the previous most probable state. The forward and backward computation phases are pipelined, therefore on the average 1/3 bits are decoded at a clock cycle.

5.7 UMTS Turbo Decoder

Turbo codes [50] are widely used in telecommunications standards as in UMTS [51] for forward error correction. The basic difference between UMTS [51] and CCSDC [52] Turbo codes in terms of implementation is the number of states

in the codes. The UMTS code uses an 8-state encoder, whereas CCSDC uses a 16-state encoder. The complexity of the decoder algorithm almost doubles when the number of states double. These two algorithms can be easily coded in LRC and mapped to the BilRC. Only the UMTS Turbo decoder will be considered, since the basic computation mechanism is the same. In Chapter 6, results will be provided for both decoders. A Turbo decoder requires an iterative decoding algorithm in which two soft-in-soft-output (SISO) decoders exchange information. The first SISO corresponds to the convolutional encoder that encodes the data in the normal order, and the second one corresponds to the encoder that encodes the data in an interleaved order. The operations performed in these two decoders are the same. Therefore, only a single decoder, which serves as both the first SISO and the second SISO sequentially, is implemented in LRC. Inside a SISO decoder, a forward recursion is performed first. At each step the probabilities of states are stored in memories and then a backward recursion is performed. During the backward recursion, the probabilities of states computed in forward recursion and the current backward state probabilities are used to compute a new likelihood ratio for the symbol to be decoded [53].

Fig. 5.7 illustrates the CDFG and LRC of a Turbo decoder. The first loop instruction (`I1`) is used to count the iterations, which starts from 0 and end at 9. The second loop (`I2`) counts SISOs. When `SISO` is 0, the instructions inside the loop body correspond to the first SISO in the algorithm. When it is 1, it behaves as the second SISO. The third loop instruction (`I3`) ,whose output is `k`, is used for forward recursion, and the loop instruction (`I4`), whose output is `m`, is used for backward recursion. The forward recursion and backward recursion instructions read the input data from the same memory. Hence, `k` and `m` are multiplexed with the `SMUX` instruction. `k` and `m` can not be active at the same time, since the loop for `m` starts after the loop for `k` exits. The input likelihoods are stored in three arrays, `syst`, `par1` and `par2` corresponding to the systematic, the parity of first encoder and the parity of second encoder, respectively. `extr` is for the extrinsic information memory. The first SISO uses `par1` as the parity likelihood, and the second SISO uses `par2`. The `EQUAL` instruction (`I12`) corresponding to `par` selects either `par1` or `par2` depending on the value of `SISO`. The arrays for `syst` and `extr`

must be accessed in the normal order for the first SISO and in the interleaved order for the second SISO. The read address of the memory, `inter_index`, is set to `k_m(2)` when SISO is 0 and `interleaver` when `SISO` is 1 by using an `EQUAL` instruction (`I9`), where `interleaver` is the interleaved address that is read from a memory.

```
[iter,iter_Exit]=FOR_SMALLER(0,20,1)<-[delay_out,SISO_Exit(1)]
    [SISO,SISO_Exit]=FOR_SMALLER(0,2,1)<-[iter,m_Exit(14)]
        [k, k_Exit ] = SFOR_SMALLER( 0,100,1,1)<-[SISO]
          [k_m,0] = SMUX(k,m)<-[]
          [par1] = MEM(0,k_m,rx_par1.txt,0,0)<-[]
          [par2] = MEM(0,k_m,rx_par2.txt,0,0)<-[]
          [interleaver]=MEM(0,k_m,interleaver.txt,0,0)<-[]
          [0,inter_index]=EQUAL(SISO,0,k_m(2),interleaver(1))
                          <-[k_m(2)]
          [syst  ] = MEM(0,inter_index,rx_sys.txt,0,0)<-[]
          [extr  ] = MEM(0,inter_index,0,wr_extr,Le2)<-[]
          [0,par ] = EQUAL(SISO,1,par2(2),par1(2))<-[par1(2)]
          ...
          #Forward Recursion Instructions
          [m, m_Exit]=SFOR_BIGGER(99,-1,-1,1)<-[k_Exit(11)]
          ...
          #Backward Recursion Instructions
```

Program 15: UMTS Turbo Decoder

Chapter 6

Results

6.1 Physical Implementation

We utilized Cadence RTL Compiler for logical synthesis and Cadence Encounter for layout generation. Faraday library[1] for 90nm UMC CMOS process technology was used for standard cells. Behavioral and gate-level simulations were performed on Cadence NC-VHDL and NC-Verilog. The steps taken in physical implementation were similar to standard ASIC implementation steps. Since BilRC has a programmable segmented-interconnect architecture, it is not possible to directly synthesize the top-level BilRC HDL code. The Cadence synthesis tool can find and optimize the critical path. Since the configuration for BilRC is unknown to the tool, it can not determine the critical path. Therefore, PEs are synthesized individually by applying two timing constraints. The combinational path delay constraint (T_{HOP}) is applied in order to determine the time delay to traverse a PE. The clock constraint is applied in order to determine the path between any PE input and the register output of the PC. The plain clock constraint is used to determine the longest delay path between two registers. Since the input of PE is not registered, this condition is specified to the tool with `-input` switch [54]. Table 6.1 shows the timing results achieved at +25°C.

[1] http://www.faraday-tech.com/index.html

Table 6.1: Timing Performance of PEs

Timing Constraint	ALU	MUL	MEM
T_{HOP}(ns)	0.188	0.188	0.188
T_{PE}(ns)	1.47	1.43	1.00

Table 6.2: Areas of PEs with 90nm UMC process

	ALU	MUL	MEM
# of cells	9823	9322	4525
Height (μ)	300	300	300
Width (μ)	240	240	400
Area (mm^2)	0.072	0.072	0.12
Layout Utilization	87	85	87

Table 6.2 shows the silicon area for PEs. The area of a PE contains both the area of the PC and the area of the PRBs. The area of the PRBs, is about 0.03mm^2. 42% of the PE area is used for PRBs in ALU and MUL and 25% for MEM. PEs were first synthesized with the Cadence RTL compiler and then placed and routed with the Cadence Encounter tool. The last row in Table 6.2 shows the percentage utilization of the rectangular area of the layout. The heights of PEs are chosen to be the same value: 300μm. However the widths are variable. Since PEs can be connected by abutment to neighboring PEs, no further area is required for interconnections. The area value for MEM contains both the area of the logic cells and the area of SRAM.

6.2 Comparison to TI C64+ DSP

Table 6.3 depicts the cycle count performance of all algorithms mapped to BilRC. The area results and the utilization of the PEs are shown in Table 6.4. The achieved clock frequencies for the applications are listed in Table 6.6. When mapping applications to BilRC, the minimum rectangular area containing a sufficient number of PEs is selected. Table 6.3 shows the cycle count performance of the applications mapped on BilRC and a TI C64+ 8 issue VLIW processor. BilRC always outperforms TI C64+ DSP. The improvements are due to adjustable parallelism in BilRC, whereas in TI C64+ the maximum number of instructions that

Table 6.3: Cycle count performance of benchmarks

Application	Notes	BilRC Cycle Count	TI C64+ Cycle Count	Ratio
2D-IDCT	100 Frames [48]	931	9262	9.95
maxval	Array size [48] 128	22	42	1.91
dotprod	Dot product, arrays size 256 [48]	41	79	1.93
maxidx	Index of maximum, array size 128 [48]	22	82	3.73
FIR	32-tap FIR filter, data size 256 [48]	266	2065	8.07
vecsum	Vector addition, size 256 [48]	36	106	2.94
FIR Complex	16-tap Complex FIR Filter, data size 256 [48]	266	4112	15.5
16-State Viterbi	Information of size 100	513	NA	NA
8-State Turbo	Chapter 5.7	8590	NA	NA
FFT	Radix-2, 1024 Point	10351	NA	NA
Multirate FIR	Rate 2, 16-tap FIR filter	1032	NA	NA
Multichannel FIR	2 channel 16-tap FIR filter	2057	NA	NA

can be executed in a single clock cycle is limited. For example, the UMTS Turbo decoder and 2D-IDCT implementations on BilRC have average instruction per cycle (IPC) values of about 30 and 128, respectively [2]. For TI's 8-issue VLIW processor, the maximum IPC is 8.

Further improvements are possible. For example, the performance of the `maxval` and `dotprod` algorithms can be doubled by storing the arrays in 16 memory blocks and processing accordingly. The performances for the TI C64+ implementations are obtained by coding these algorithms in the assembly language. Obtaining these performances is quite difficult and requires considerable expertise in the specific assembly language for the targeted VLIW processor. Table 6.4 and Table 6.6 show the area and timing results for BilRC. Although TMS320C64 has a faster clock of 1000 MHz, BilRC provides better throughput results (except for the `maxval` and `dotprod` algorithms). The TMS320C64's processor core area is reported to be $2mm^2$ [22], while the whole chip area, including two level caches and peripherals, is $20mm^2$. As is clear from Table 6.7, all of the applications mapped on BilRC requires an area of less than $20mm^2$ (except the `FIR Complex` algorithm). If the primary concern in regard to implementing an application is the area, the parallelism degree can be decreased to fit the given area. For example, the area of the `FIR Complex` can be reduced to a quarter of the value indicated by performing complex multiplication operations in the algorithm sequentially. BilRC and its computation model allow the designer to balance the area and performance.

6.3 Comparison to Xilinx Virtex-4 FPGA

One of the main advantages of CGRA as compared to FPGAs is the reduction in the configuration size. This reduction allows CGRA to be configured at run time. For a comparison of configuration size, Xilinx Virtex4 FPGA is used. This FPGA is partitioned into four rows. Inside a row, 16 configurable logic blocks (CLB) form a column. Similarly, there are four BRAMs and eight DSP48 blocks in a column. The resources forming a column are configured together. Table 6.5

Table 6.4: Comparison of configuration sizes of BilRC and Xilinx Virtex4

	BilRC					Xilinx Virtex4			FPGA/BilRC Conf Ratio
Application	# of PEs (ALU, MUL, MEM)	Rows-Columns	Area (mm^2)	Utilization (%)	Conf. Bits	Columns (CLB, DSP, BRAM)	Conf Bits	Utilization (%) (SLICEM, SLICEL, DSP, LUT, BRAM)	
idct	114,38,8	16-14	17.7	71	39552	32,4,4	1138816	91,89,100,58,50	28.8
maxval	17,0,8	8-4	2.5	78	6016	6,0,2	225664	71,71,NA,71,100	37.5
dotprod	32,16,16	16-5	6.5	80	14336	16,4,4	676992	58,57,25,40,100	47.2
maxidx	17,0,8	8-4	2.5	78	5760	6,0,2	225664	70,70,NA,70,100	39.2
FIR	33,32,1	8-16	9.98	52	22528	16,4,4	676992	92,92,100,28,7	30.1
vecsum	10,0,24	24-3	6.3	47	12672	6,0,6	330624	39,39,NA,25,100	26.1
FIR Complex	128,128,2	32-16	39.9	50	90112	40,8,8	1584896	96,96,100,41,7	17.6
16-State Viterbi	76,3,3	11-11	9.7	68	22096	15,1,1	486752	94,93,0,93,75	22
UMTS Turbo	107,0,13	14-11	11.7	78	27904	24,0,4	797696	85,84,NA,83,82	28.6
FFT	25,9,5	5-9	3.67	87	8016	8,2,2	338496	86,85,25,49,63	42.2
multirate FIR	17,16,1	8-7	4.4	61	9856	10,2,2	396224	98,98,100,23,13	40.2
multichannel FIR	(18,16,2)	8-7	4.4	61	9856	12,2,2	396224	86,86,100,24,23	46.1
Arithmetic Mean				68				80,81,69,60,50	33.8

shows the number of frames required to configure different column types [55]. A configuration frame is composed of 1312 bits. For CLB and DSP48 (the multiplier block), the configuration stream configures both the functionality of the blocks in the column and the interconnection network. The configuration stream for BRAM initialization and interconnect is separately provided [55].

To make a fair configuration size comparison, only the required number of configuration columns should be taken into account. This is done by using the Xilinx PlanAhead tool which allows all resources (CLB, DSP48, BRAM) to be placed and routed within a *partition block* (PBlock). When drawing a PBlock, the height must be at a row boundary since the resources in a column are configured together. The width of the PBlock, on the other hand, must be selected so that enough resources exist in the PBlock.

HDL code generated from the LRC-HDL converter is used as the input to the Xilinx ISE tool. When mapping the applications to the FPGA, the locations of the PBlocks are manually selected to increase the utilization of resources to reduce configuration size. When mapping the applications to BilRC, a minimum-sized rectangle, starting from the top-left PE, is formed containing sufficient resources (ALU, MEM, MUL). The BilRC placement and routing tool places PEs in the selected rectangle. Only the interconnect resources within the selected rectangle area are used for signal routing. The tool is forced to use only three ports per PE side ($N_p = 3$), and all applications are routed without congestion. Although three ports are enough for the selected applications, all performance results (configuration size, area and timing) are given for $N_p = 4$, leaving extra flexibility for more complex applications. The results are summarized in Table 6.4. For example, the FFT algorithm requires 39 PEs arranged in nine rows and five columns with an utilization ratio of 87% and it can be configured with just 8016 bits[2]. To implement the same algorithm, Virtex4 requires 8 CLBs, 2 DSP and 2 BRAM columns configured with 338,496 bits. Utilizations of various logic resources are shown in the ninth column of the table. The last column lists the improvements in the configuration size varying from 17.6× to 47.2×.

[2]This number includes the configuration bits for unused PEs.

Table 6.5: Configuration Frames for FPGA Resources

Column Type	CLB	BRAM inter-connect	BRAM content	DSP48
# of frames	22	20	64	21

CGRAs are expected to provide better timing performance as compared to FPGAs. The arithmetic units of a CGRA are pre-placed and routed, whereas in an FPGA, these units are formed from LUTs. The critical path for an instruction in a CGRA is formed from gates that are, in general, faster than LUTs. In [56] the gap between FPGA and ASIC implementations are measured, it is found that ASICs are on the average three times faster than FPGA implementations. This value is found by allowing the use of the hard blocks (multiplier and memory) during algorithm mapping to an FPGA. Since CGRAs cannot be faster than ASICs, a well-designed CGRA is at best three times faster than an FPGA. Table 6.6 shows the critical path delays of BilRC and Xilinx Virtex4 implemented with the same 90nm CMOS technology. The second column shows the worst case hop count between a source PE and a destination PE. The critical path of PEs is taken as 1.47ns, which is the worst performance among PEs. Improvements in the range of 1.53× and 3.6× are obtained.

6.4 Comparison to other CGRAs

The 2D-IDCT algorithm has been implemented on many CGRAs. The results are shown in Table 6.7. In terms of cycle count, BilRC is 3.2 times faster than the fastest CGRA, ADRES [22]. In terms of throughput, BilRC is 2.2 times faster than ADRES. The maximum clock frequency of BilRC for IDCT algorithm is found to be 415 MHz. ADRES and MORA work at a constant frequency of 600 and 1000 MHz respectively. The timing result of MorphoSys is not available for 90nm technology, and its area result is scaled to 90nm in the table. The lower operating frequency of BilRC is due to its segmented interconnect network. BilRC uses a larger silicon area for implementing the IDCT algorithm, mainly

Table 6.6: Critical Path Comparison of BilRC and FPGA

Application	# of Hops	BilRC Clock (MHz)	Virtex4 Clock (MHz)	Speedup
idct	5	415	147	2.82
maxval	4	450	251	1.79
dotprod	4	450	125	3.6
maxidx	4	450	244	1.84
FIR	3	492	174	2.82
vecsum	4	450	247	1.82
FIR Complex	4	450	145	3.1
16-State Viterbi	5	415	204	2.03
8-State Turbo	6	385	251	1.53
FFT	3	492	147	3.34
Multirate FIR	3	492	152	3.23
Multichannel FIR	3	492	167	2.94
Arithmetic Mean				2.57

Table 6.7: Area, Timing and Cycle Count Results for the 2D-IDCT Algorithm

CGRA	# of PEs	Area (mm^2)	Granularity	Average Cycle Count	Clock Freq. (MHz)	Throughput (Million IDCT/sec)
BilRC	152	11.90	16-bit	9.3	415	44.6
ADRES	64	4	32-bit	30	600	20
MORA	22	1.749	8-bit	108	1000	10.2
MorphoSys	64	11.11	16-bit	37	NA	NA

Table 6.8: IPC and Scheduling Density Comparison

	FFT		IDCT	
	IPC	SD	IPC	SD
BilRC	17.8	54%	128	85%
ADRES[18]	23.3	37%	31(V),42(H)	45%(V),47%(H)
ADRES[58]	10.4	65%	NA	NA
ADRES[59]	12.4	78%	13.3	83%

due to its flexible segmented interconnect architecture which is crucial for the high performance implementation of a broad range of applications. The area result for MorphoSys includes the area for a small RISC processor and some other peripherals. It was reported that more than 80% of the whole chip area was used for the reconfigurable arrays [57]. The area result for ADRES includes the area of the VLIW processor as well.

BilRC does not require an external processor for loop control or execution control, however an external processor can be attached to BilRC for the execution of sequential code for initializations and parameter loading.

ADRES processor is a mature CGRA. ADRES has the significant advantage of mapping full applications from the C language, a property that BilRC does not yet have.

In BilRC, PEs are statically configured, whereas the reported CGRAs rely on dynamic reconfiguration. In general, dynamically reconfigurable CGRAs are expected to provide better PE utilization. However, due to its execution-triggered computation model and flexible interconnect architecture, BilRC provides better or comparable PE utilization. For example, BilRC requires 152 PEs for the IDCT algorithm with an average IPC (instruction per cycle) of about 128 [2]. Therefore, the scheduling density is about 85%, whereas ADRES [18] has scheduling densities (SD) of 45% for the vertical phase of IDCT (V) and 66% for the horizontal phase of IDCT (H). Table 6.8 compares BilRC with 3 ADRES implementations.

Chapter 7

Conclusion

We have presented BilRC and its LRC language, capable of implementing state of the art algorithms with very good performance in speed, area utilization, and configuration size. BilRC contains three different kinds of PEs. Using 90nm technology, 14 16-bit PEs can fit into $1mm^2$ of silicon. The total number of PEs is equal to the number of instructions in LRC code. The FFT algorithm can be implemented with just 39 instructions.

The reduction in configuration size is possible mainly for two reasons. First, 17-bit signals are routed together in BilRC, whereas in an FPGA each bit is individually routed. Second, the functionality of a PE is selected with an 8-bit opcode, whereas in an FPGA functionality is programmed by filling in several look-up-tables (LUT). The configuration size, area and timing performance can be further improved by optimizing the interconnect architecture.

BilRC can be used as an accelerator attached to a DSP processor for applications requiring high computation power. Due to the run-time configurability of BilRC, several applications can be run in a time-multiplexed manner. BilRC may also be used as an alternative to FPGAs, especially for applications having word level granularity. Almost all telecommunications and signal processing algorithms have word-level granularity. The main advantages of BilRC as compared to FPGAs are run-time configurability due to reduced configuration size, reduced

compilation time and faster frequency of operation.

Bibliography

[1] M. Nicola, G. Masera, M. Zamboni, H. Hishebabi, D. Kammler, G. Ascheid, and H. Meyr, "FFT processor: a case study in asip development," in *IST Mobile Summit*, 2005.

[2] O. Atak and A. Atalar, "An efficient computation model for coarse grained reconfigurable architectures and its applications to a reconfigurable computer," in *Application-specific Systems Architectures and Processors (ASAP), 21st IEEE International Conference on*, pp. 289 –292, july 2010.

[3] H. Ishebabi, G. Ascheid, H. Meyr, O. Atak, A. Atalar, and E. Arikan, "An efficient parallelization technique for high throughput fft-asips," in *Circuits and Systems, 2006. ISCAS 2006. Proceedings. 2006 IEEE International Symposium on*, p. 4 pp., may 2006.

[4] X. Guan, Y. Fei, and H. Lin, "Hierarchical design of an application-specific instruction set processor for high-throughput and scalable fft processing," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 20, pp. 551 –563, march 2012.

[5] T. Vogt and N. Wehn, "A reconfigurable ASIP for convolutional and Turbo decoding in an SDR environment," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, pp. 1309 –1320, oct. 2008.

[6] O. Muller, A. Baghdadi, and M. Jezequel, "From parallelism levels to a multi-ASIP architecture for Turbo decoding," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 92 –102, jan. 2009.

[7] K. Karuri, A. Chattopadhyay, X. Chen, D. Kammler, L. Hao, R. Leupers, H. Meyr, and G. Ascheid, "A design flow for architecture exploration and implementation of partially reconfigurable processors," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 16, pp. 1281 –1294, oct. 2008.

[8] R. Hartenstein, "A decade of reconfigurable computing: A visionary perspective," in *Proceedings of European Design, Automation and Test Conference, (DATE)*, 2001.

[9] B. De Sutter, P. Raghavan, and A. Lambrechts, "Coarse-grained reconfigurable array architectures," in *Handbook of Signal Processing Systems* (S. S. Bhattacharyya, E. F. Deprettere, R. Leupers, and J. Takala, eds.), pp. 449–484, Springer US, 2010.

[10] K. Compton and S. Hauck, "Reconfigurable computing: A survey of systems and software," *ACM Computing Surveys*, vol. 34, no. 2, pp. 171–210, 2002.

[11] Y. Kim and R. Mahapatra, "Dynamic context compression for low-power coarse-grained reconfigurable architecture," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 18, pp. 15 –28, jan. 2010.

[12] Y. Kim, R. Mahapatra, I. Park, and K. Choi, "Low power reconfiguration technique for coarse-grained reconfigurable architecture," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. 17, pp. 593 –603, may 2009.

[13] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor, "PipeRench: A reconfigurable architecture and compiler," *IEEE Computer*, vol. 33, no. 4, pp. 70–77, 2000.

[14] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. Taylor, and R. Laufer, "Piperench: a coprocessor for streaming multimedia acceleration," in *Computer Architecture, 1999. Proceedings of the 26th International Symposium on*, pp. 28 –39, 1999.

[15] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. Reed Taylor, "Piperench: A virtualized programmable datapath in 0.18 micron technology," in *Custom Integrated Circuits Conference, 2002. Proceedings of the IEEE 2002*, pp. 63 – 66, 2002.

[16] C. Ebeling, C. Fisher, G. Xing, M. Shen, and H. Liu, "Implementing an OFDM receiver on the RaPiD reconfigurable architecture," *Computers, IEEE Transactions on*, vol. 53, pp. 1436 – 1448, nov. 2004.

[17] C. Ebeling, D. Cronquist, and P. Franklin, "RaPiD reconfigurable pipelined datapath," in *Field-Programmable Logic Smart Applications, New Paradigms and Compilers* (R. Hartenstein and M. Glesner, eds.), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[18] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," *Computers and Digital Techniques, IEE Proceedings* -, vol. 150, pp. 255–61, sept. 2003.

[19] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "ADRES: An architecture with tightly coupled VLIW processor and coarse-grained reconfigurable matrix," in *Field Programmable Logic and Application* (P. Y. K. Cheung and G. Constantinides, eds.), vol. 2778 of *Lecture Notes in Computer Science*, pp. 61–70, Springer Berlin / Heidelberg, 2003.

[20] B. Mei, A. Lambrechts, J.-Y. Mignolet, D. Verkest, and R. Lauwereins, "Architecture exploration for a reconfigurable architecture template," *Design Test of Computers, IEEE*, vol. 22, pp. 90 – 101, march-april 2005.

[21] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the ADRES coarse-grained reconfigurable array," in *Reconfigurable Computing: Architectures, Tools and Applications* (P. Diniz, E. Marques, K. Bertels, M. Fernandes, and J. Cardoso, eds.), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[22] M. Berekovic, A. Kanstein, B. Mei, and B. De Sutter, "Mapping of nomadic multimedia applications on the adres reconfigurable array processor," *Microprocess. Microsyst.*, vol. 33, June 2009.

[23] H. Singh, M.-H. Lee, G. Lu, F. Kurdahi, N. Bagherzadeh, and E. Chaves Filho, "MorphoSys: an integrated reconfigurable system for data-parallel and computation-intensive applications," *Computers, IEEE Transactions on*, vol. 49, pp. 465 –481, may 2000.

[24] M. Lanuzza, S. Perri, P. Corsonello, and M. Margala, "Energy efficient coarse-grain reconfigurable array for accelerating digital signal processing," in *Integrated Circuit and System Design. Power and Timing Modeling, Optimization and Simulation* (L. Svensson and J. Monteiro, eds.), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[25] W. Vanderbauwhede, M. Margala, S. Chalamalasetti, and S. Purohit, "Programming model and low-level language for a coarse-grained reconfigurable multimedia processor," in *Proceedings of the 2009 International Conference on Engineering of Reconfigurable Systems and Algorithms, ERSA 2009, July 13 - 16, 2009, Las Vegas, Nevada, USA* (T. Plaks, ed.), CSREA.

[26] T. Miyamori and K. Olukotun, "Remarc: reconfigurable multimedia array coprocessor," in *Proceedings of the 1998 ACM/SIGDA sixth international symposium on Field programmable gate arrays*, FPGA '98, 1998.

[27] J. Hauser and J. Wawrzynek, "Garp: a MIPS processor with a reconfigurable coprocessor," in *FPGAs for Custom Computing Machines, 1997. Proceedings., The 5th Annual IEEE Symposium on*, pp. 12 –21, apr 1997.

[28] T. Callahan, J. Hauser, and J. Wawrzynek, "The garp architecture and c compiler," *Computer*, vol. 33, pp. 62 –69, apr 2000.

[29] R. Hartenstein and R. Kress, "A datapath synthesis system for the reconfigurable datapath architecture," in *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International*

Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific, pp. 479 –484, aug-1 sep 1995.

[30] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger, "Kressarray xplorer: a new cad environment to optimize reconfigurable datapath array architectures," in *Design Automation Conference, 2000. Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pp. 163 –168, june 2000.

[31] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to software: RAW machines," *Computer*, vol. 30, pp. 86 –93, sep 1997.

[32] E. Mirsky and A. DeHon, "Matrix: a reconfigurable computing architecture with configurable instruction distribution and deployable resources," in *FPGAs for Custom Computing Machines, 1996. Proceedings. IEEE Symposium on*, pp. 157 –166, apr 1996.

[33] R. Bittner, P. M. Athanas, and M. Musgrove, "Colt: an experiment in wormhole run-time reconfiguration," in *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series* (J. Schewel, P. M. Athanas, V. M. Bove, and J. Watson, eds.), vol. 2914 of *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, pp. 187–194, Oct. 1996.

[34] J. Cardoso and M. Weinhardt, "Xpp-vc: A c compiler with temporal partitioning for the pact-xpp architecture," in *Field-Programmable Logic and Applications: Reconfigurable Computing Is Going Mainstream* (M. Glesner, P. Zipf, and M. Renovell, eds.), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[35] "Pact xpp technologies: Xpp-iii processor overview white paper,"

[36] V. Baumgarte, G. Ehlers, F. May, A. Nckel, M. Vorbach, and M. Weinhardt, "Pact xppa self-reconfigurable data processing architecture," *The Journal of Supercomputing*, vol. 26, pp. 167–184.

[37] R. Hartenstein, M. Herz, , T. Hoffmann, and U. Nageldinger, "Kressarray Xplorer: a new CAD environment to optimizere configurable datapath array architectures," in *Proc. ASP-Design Automation Conference*, pp. 163–168, Jan. 2000.

[38] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Dresc: a retargetable compiler for coarse-grained reconfigurable architectures," in *Field-Programmable Technology, 2002. (FPT). Proceedings. 2002 IEEE International Conference on*, pp. 166 – 173, dec. 2002.

[39] J. M. P. Cardoso and P. C. Diniz, *Compilation Techniques for Reconfigurable Architectures*. Springer Publishing Company, Incorporated, 1st ed., 2010.

[40] J. Hammes, B. Rinker, W. Bohm, W. Najjar, B. Draper, and R. Beveridge, "Cameron: high level language compilation for reconfigurable systems," in *Parallel Architectures and Compilation Techniques, 1999. Proceedings. 1999 International Conference on*, pp. 236 –244, 1999.

[41] W. Bhm, J. Hammes, B. Draper, M. Chawathe, C. Ross, R. Rinker, and W. Najjar, "Mapping a single assignment programming language to reconfigurable systems," *The Journal of Supercomputing*, vol. 21, pp. 117–130, 2002. 10.1023/A:1013623303037.

[42] A. H. Veen, "Dataflow machine architecture," *ACM Comput. Surv.*, vol. 18, pp. 365–396, December 1986.

[43] C. Jang, J. Kim, J. Lee, H.-S. Kim, D.-H. Yoo, S. Kim, H.-S. Kim, and S. Ryu, "An instruction-scheduling-aware data partitioning technique for coarse-grained reconfigurable architectures," in *Proceedings of the 2011 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems*, LCTES '11, pp. 151–160, 2011.

[44] B. De Sutter, O. Allam, P. Raghavan, R. Vandebriel, H. Cappelle, T. Vander Aa, and B. Mei, "An efficient memory organization for high-ILP inner modem baseband SDR processors," *J. Signal Process. Syst.*, vol. 61, pp. 157–179, Nov. 2010.

[45] M. Lam, "Software pipelining: an effective scheduling technique for vliw machines," *SIGPLAN Not.*, vol. 23, pp. 318–328, June 1988.

[46] V. Betz and J. Rose, "VPR: a new packing, placement and routing tool for fpga research," in *Field-Programmable Logic and Applications* (W. Luk, P. Cheung, and M. Glesner, eds.), Lecture Notes in Computer Science, Springer Berlin / Heidelberg.

[47] L. McMurchie and C. Ebeling, "Pathfinder: A negotiation-based performance-driven router for FPGAs," in *Field-Programmable Gate Arrays, 1995. FPGA '95. Proceedings of the Third International ACM Symposium on.*

[48] "Texas instruments inc. TMS320C674x low power DSPs." `http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip`.

[49] A. Viterbi, "An intuitive justification and a simplified implementation of the MAP decoder for convolutional codes," *Selected Areas in Communications, IEEE Journal on*, vol. 16, pp. 260 –264, feb 1998.

[50] C. Berrou, A. Glavieux, and P. Thitimajshima, "Near Shannon limit error-correcting coding and decoding: Turbo-codes. 1," in *Communications, 1993. ICC 93. Geneva. Technical Program, Conference Record, IEEE International Conference on.*

[51] "European telecommunications standards institute, universal mobile telecommunications system (UMTS): Multiplexing and channel coding (FDD)," in *3GPP TS 125.212 version 3.4.0*, 2000.

[52] "TM Synchronization and Channel Coding," in *CCSDC 130.1-G-1*, 2006.

[53] M. C. Valenti and J. Sun, "The UMTS turbo code and an efficient decoder implementation suitable for software-defined radios," *International Journal of Wireless Information Networks*, vol. 8, pp. 203–215, 2001.

[54] "Synopsys timing constraints and optimization user guide," in *Version C-2009.06.*

[55]

[56] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, pp. 203 –215, feb. 2007.

[57] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. Filho, and V. C. Alves, "Design and implementation of the MorphoSys reconfigurable computing processor," *The Journal of VLSI Signal Processing*.

[58] B. Bougard, B. De Sutter, S. Rabou, D. Novo, O. Allam, S. Dupont, and L. Van der Perre, "A coarse-grained array based baseband processor for 100mbps+ software defined radio," in *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, (New York, NY, USA), pp. 716–721, ACM, 2008.

[59] B. De Sutter, P. Coene, T. Vander Aa, and B. Mei, "Placement-and-routing-based register allocation for coarse-grained reconfigurable arrays," in *Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '08, (New York, NY, USA), pp. 151–160, ACM, 2008.

Appendix A

Acronyms

ASIC Application Specific Integrated Circuit

ASIP Application Specific Instruction-set Processors

ALU Arithmetic Logic Unit

BRAM Block Random Access Memory

BilRC Bilkent Reconfigurable Computer

CDFG Control Data Flow Graphs

CGRA Coarse Grained Reconfigurable Architecture

CLB Configurable Logic Block

CMOS Complementary Metal Oxide Semiconductor

CPU Central Processing Unit

CR Configuration Register

DCT Discrete Cosine Transform

DSP Digital Signal Processor

EE Execute Enable

FIR Finite Impulse Response

FFT Fast Fourier Transform

FPGA Field Programmable Gate Array

HDL Hardware Description Language

IDCT Inverse Discrete Cosine Transform

IPC Instruction Per Cycle

LDPC Low Density Parity Check

LRC a Language for Reconfigurable Computing

LTE Long Term Evolution

MemID Memory Identification

p2p Point to Point

PC Processing Core

PE Processing Element

PRB Port Route Box

RA Reconfigurable Architectures

RAM Random Access Memory

RC Reconfigurable Computer

RISC Reduced Instruction Set Computer

SRAM Static Random Access Memory

VCD Value Change Dump

VLIW Very Long Instruction Word

WIMAX Worldwide Interoperability for Microwave Access

Appendix B

Instruction Set Of BilRC

B.1 ABS

Syntax: `[result(init)]=ABS(A)<-[EE,init_EE]`
Description:Calculates the absolute value of `A` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.2 ADD

Syntax: `[result(init),carry]=ADD(A,B)<-[EE,init_EE]`
Description:Calculates the sum of two operands `A` and `B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.3 ADD_MM

Syntax: `[result(init),carry]=ADD_MM(A,B)<-[EE,init_EE]`
Description: Sign inverts `A` and `B` and then calculates the sum of `-A` and `-B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is

active.

B.4 ADD_C

Syntax: `[result(init),carry]=ADD(A,B,carry_in)<-[EE,init_EE]`
Description:Calculates the sum of three operands `A`, `B` and `carry_in` when `EE` is active. It must be noted that only the least significant bit of the `carry_in` is taken into account. The output variable `result` is assigned to `init` when `init_EE` is active.

B.5 AND

Syntax: `[result(init)]=AND(A,B)<-[EE,init_EE]`
Description:Calculates the logical AND of two operands `A` and `B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.6 BIGGER

Syntax: `[result(init),cond_result]=BIGGER(A,B,C,D)<-[EE,init_EE]`
Description: The instruction is executed only when `EE` is active. `result` is assigned to `C` if `A>B` otherwise it is assigned to `D`. `cond_result` is assigned to `C` only if `A>B`. The Execute Enable part of `cond_result` is activated only if `A>B`. The output variable `result` is assigned to `init` when `init_EE` is active.

B.7 DELAY

Syntax: `[result(init)]=DELAY(A)<-[EE,init_EE]`
Description: `result` is assigned to `A` when `EE` is active. The output variable

result is assigned to init when init_EE is active.

B.8 EQUAL

Syntax: [result(init),cond_result]=EQUAL(A,B,C,D)<-[EE,init_EE]
Description: The instruction is executed only when EE is active. result is assigned to C if A is equal to B otherwise it is assigned to D. cond_result is assigned to C only if A is equal to B. The Execute Enable part of cond_result is activated only if A is equal to B. The output variable result is assigned to init when init_EE is active.

B.9 FOR_BIGGER

Syntax: [i,i_Exit]=FOR_BIGGER(Start,Final,Incr)<-[LoopStart,Next]
Description: It corresponds to the for loop in C language with syntax for(i=Start,i>Final,i=i+Incr). Detailed explanation can be found in Chapter 3.3.1

B.10 FOR_SMALLER

Syntax: [i,i_Exit]=FOR_SMALLER(Start,Final,Incr)<-[LoopStart,Next]
Description: It corresponds to the for loop in C language with syntax for(i=Start,i<Final,i=i+Incr). Detailed explanation can be found in Chapter 3.3.1

B.11 MAX

Syntax: `[result(init),IndexOut]=MAX(A,IndexA,B,IndexB)<-[EE,init_EE]`
Description: `result` is assigned to maximum of `A` and `B` when `EE` is active. The second output `IndexOut` is assigned `IndexA` if `A` is the maximum, otherwise it is assigned to `B`. The first output `result` is initialized to `init` when `init_EE` is active.

B.12 MAX

Syntax: `[result(init),IndexOut]=MAX(A,IndexA,B,IndexB)<-[EE,init_EE]`
Description: `result` is assigned to maximum of `A` and `B` when `EE` is active. The second output `IndexOut` is assigned `IndexA` if `A` is the maximum, otherwise it is assigned to `B`. The first output `result` is initialized to `init` when `init_EE` is active.

B.13 MERGE

Syntax: `[result(init)]=MERGE(BitWidth,A,B,C,D)<-[EE,init_EE]`
Description: Below description is in VHDL notation. It is assumed that all signals are of type `STD_LOGIC_VECTOR`.

```
if BitWidth=x"0000" then
    result(3 downto 0)<=A(0)&B(0)&C(0)&D(0);
elsif BitWidth=x"0001" then
    result(7 downto 0)<= A(1 downto 0)
                         &B(1 downto 0)
                         &C(1 downto 0)
                         &D(1 downto 0);
elsif BitWidth=x"0002" then
```

```
        result(15 downto 0)<=A(3 downto 0)
                             &B(3 downto 0)
                             &C(3 downto 0)
                             &D(3 downto 0);
    elsif BitWidth=x"0003" then
        result(15 downto 0)<=A(7 downto 0)
                             &B(7 downto 0);
    end if;
```

B.14 MUL_SHIFT

Syntax: `[result_lsb(init),result_msb]=MUL_SHIFT(A,B,C)<-[EE,init_EE]`
Description: `A` and `B` multiplied and the result is shifted to the right by `C` and then the 16 least significant bits of the result is assigned to `result_lsb` and 16 most significant bits are assigned to `result_msb`.

B.15 MULTIPLEX

Syntax: `[result(init)]=MULTIPLEX(sel,A,B,C,D)<-[EE,init_EE]`
Description: `result` is assigned to `A`, `B`, `C` or `D` if `sel` is 0, 1, 2 or 3, respectively. The instruction is executed only when `EE` is active. `result` is initialized to `init` when `init_EE` is active.

B.16 NOT

Syntax: `[result(init)]=NOT(A)<-[EE,init_EE]`
Description: `result` is assigned to logical complement of `A` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.17 NOT_EQUAL

Syntax: `[result(init),cond_result]=NOT_EQUAL(A,B,C,D)<-[EE,init_EE]`
Description: The instruction is executed only when `EE` is active. `result` is assigned to `C` if `A` is not equal to `B` otherwise it is assigned to `D`. `cond_result` is assigned to `C` only if `A` is not equal to `B`. The Execute Enable part of `cond_result` is activated only if `A` is not equal to `B`. The output variable `result` is assigned to `init` when `init_EE` is active.

B.18 OR

Syntax: `[result(init)]=OR(A,B)<-[EE,init_EE]`
Description:Calculates the logical OR of two operands `A` and `B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.19 SAT

Syntax: `[result(init)]=SAT(A,B,C)<-[EE,init_EE]`
Description:`result` is assigned to `A`, if `A` is greater than `B` and smaller than `C`. If `A` is greater than `C`, it is assigned to `C`, i.e., saturated to `C`. If `A` is smaller than `B`, it is assigned to `B`. The instruction is executed only when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.20 SMUX

Syntax: `[result]=SMUX(A,B,C,D)<-[]`
Description:`result` is assigned to one of the operands with active Execute Enable part. `A` has the highest priority while `D` has the lowest. If more than one operand is active in a cycle, the operand with higher priority is assigned to `result`.

B.21 SFOR_BIGGER

Syntax: `[i,i_Exit]=SFOR_BIGGER(Start,Final,Incr,IID)<-[LoopStart]`
Description: It corresponds to the for loop in `C` language with syntax `for(i=Start,i>Final,i=i+Incr)`. Detailed explanation can be found in Chapter 3.3.1

B.22 SFOR_SMALLER

Syntax: `[i,i_Exit]=SFOR_SMALLER(Start,Final,Incr,IID)<-[LoopStart]`
Description: It corresponds to the for loop in `C` language with syntax `for(i=Start,i<Final,i=i+Incr)`. Detailed explanation can be found in Chapter 3.3.1

B.23 SHL_AND

Syntax: `[result(init)]=SHL_AND(A,B,C)<-[EE,init_EE]`
Description:`A` is shifted to the left by `B` and logically ANDed with `C`. The instruction is executed only when `EE` is active. `result` is initialized to `init` when `init_EE` is active.

B.24 SHL_OR

Syntax: `[result(init)]=SHL_OR(A,B,C)<-[EE,init_EE]`
Description:`A` is shifted to the left by `B` and logically ORed with `C`. The instruction is executed only when `EE` is active. `result` is initialized to `init` when `init_EE` is active.

B.25 SHR_AND

Syntax: `[result(init)]=SHR_AND(A,B,C)<-[EE,init_EE]`
Description:`A` is shifted to the right by `B` and logically ANDed with `C`. The instruction is executed only when `EE` is active. `result` is initialized to `init` when `init_EE` is active.

B.26 SHR_OR

Syntax: `[result(init)]=SHR_OR(A,B,C)<-[EE,init_EE]`
Description:`A` is shifted to the right by `B` and logically ORed with `C`. The instruction is executed only when `EE` is active. `result` is initialized to `init` when `init_EE` is active.

B.27 SMALLER

Syntax: `[result(init),cond_result]=SMALLER(A,B,C,D)<-[EE,init_EE]`
Description: The instruction is executed only when `EE` is active. `result` is assigned to `C` if `A<B` otherwise it is assigned to `D`. `cond_result` is assigned to `C` only if `A<B`. The Execute Enable part of `cond_result` is activated only if `A<B`. The output variable `result` is assigned to `init` when `init_EE` is active.

B.28 SUB

Syntax: `[result(init),carry]=SUB(A,B)<-[EE,init_EE]`
Description:Calculates the difference of two operands `A` and `B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

B.29 XOR

Syntax: `[result(init)]=XOR(A,B)<-[EE,init_EE]`

Description:Calculates the logical XOR of two operands `A` and `B` when `EE` is active. The output variable `result` is assigned to `init` when `init_EE` is active.

Appendix C

LRC code of the Algorithms

C.1 2D IDCT Algorithm

```
#####################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
#####################################################################
%PI:INPUT
%ver_f0s:OUTPUT
%ver_f1s:OUTPUT
%ver_f2s:OUTPUT
%ver_f3s:OUTPUT
%ver_f4s:OUTPUT
%ver_f5s:OUTPUT
%ver_f6s:OUTPUT
%ver_f7s:OUTPUT
#Primary Input(PI) pulse to initiate execution
[delay_out, 0 ]=DELAY(PI)<-[PI]
#There are 100 frames each of which is 8x8 blocks.
[frame_cnt, frame_Exit ] = SFOR_SMALLER( 0,100,1,8)<-[delay_out]
 #The loop below is for horizontal line counting
 [hor_line_cnt,hor_line_Exit]=SFOR_SMALLER(0,8,1,0)<-[frame_cnt]
  #the memory address is calculated from frame counter and
  #horizontal line counter
  [hor_line_adres]=SHL_OR(frame_cnt,3,hor_line_cnt)<-[hor_line_cnt]
  #8 input data are read from the memory
  [data_1] = MEM(0,hor_line_adres(2),data_1,0,0)<-[]#1
  [data_2] = MEM(0,hor_line_adres(2),data_2,0,0)<-[]
  [data_3] = MEM(0,hor_line_adres(2),data_3,0,0)<-[]
  [data_4] = MEM(0,hor_line_adres(2),data_4,0,0)<-[]
```

```
[data_5] = MEM(0,hor_line_adres(2),data_5,0,0)<-[]
[data_6] = MEM(0,hor_line_adres(2),data_6,0,0)<-[]
[data_7] = MEM(0,hor_line_adres(2),data_7,0,0)<-[]
[data_8] = MEM(0,hor_line_adres(2),data_8,0,0)<-[]
#[16384 4520 8867 12873  16384 19266  21407  22725]
# C_set1=[C4 C7 C6 C5 C4 C3 C2 C1];
[mul_1_0]  = MUL_SHIFT(data_1,16384,15)<-[data_1]#2
[mul_1_1]  = MUL_SHIFT(data_2,4520,15) <-[data_2]
[mul_1_2]  = MUL_SHIFT(data_3,8867,15) <-[data_3]
[mul_1_3]  = MUL_SHIFT(data_4,12873,15)<-[data_4]
[mul_1_4]  = MUL_SHIFT(data_5,16384,15)<-[data_5]
[mul_1_5]  = MUL_SHIFT(data_6,19266,15)<-[data_6]
[mul_1_6]  = MUL_SHIFT(data_7,21407,15)<-[data_7]
[mul_1_7]  = MUL_SHIFT(data_8,22725,15)<-[data_8]
#[16384 22725 21407  19266  16384  12873  8867   4520]
#C_set2=[C4 C1 C2 C3 C4 C5 C6 C7];
[mul_2_0]  = MUL_SHIFT(data_1,16384,15)<-[data_1]
[mul_2_1]  = MUL_SHIFT(data_2,22725,15)<-[data_2]
[mul_2_2]  = MUL_SHIFT(data_3,21407,15)<-[data_3]
[mul_2_3]  = MUL_SHIFT(data_4,19266,15)<-[data_4]
[mul_2_4]  = MUL_SHIFT(data_5,16384,15)<-[data_5]
[mul_2_5]  = MUL_SHIFT(data_6,12873,15)<-[data_6]
[mul_2_6]  = MUL_SHIFT(data_7,8867,15)<-[data_7]
[mul_2_7]  = MUL_SHIFT(data_8,4520,15)<-[data_8]
[Q1] = SUB(mul_1_1, mul_1_7)<-[mul_1_7]#3
[Q0] = SUB(mul_1_5, mul_1_3)<-[mul_1_5]#3
[S0] = ADD(mul_2_5, mul_2_3)<-[mul_1_5]#3
[S1] = ADD(mul_2_1, mul_2_7)<-[mul_2_7]#3
[p0] = ADD(mul_1_0 , mul_1_4)<-[mul_1_4]#3
[p1] = SUB(mul_2_0 , mul_2_4)<-[mul_2_4]#3
[r1] = SUB(mul_1_2 , mul_1_6)<-[mul_1_6]#3
[r0] = ADD(mul_2_2 , mul_2_6)<-[mul_2_6]#3
[ss1] = ADD(S1, S0)<-[S1]#4
[qq1] = ADD(Q1, Q0)<-[Q1]#4
[ss0] = SUB(S1, S0)<-[S1]#4
[qq0] = SUB(Q1, Q0)<-[Q1]#4
[g0] = ADD(p0, r0)<-[r0]#4
[g1] = ADD(p1, r1)<-[r1]#4
[h0] = SUB(p0, r0)<-[r0]#4
[h1] = SUB(p1, r1)<-[r1]#4
[mulA]=MUL_SHIFT(ss0,23170,15)<-[ss0]#5
[mulB]=MUL_SHIFT(qq0,23170,15)<-[qq0]#5
[g3] = SUB(mulA , mulB)<-[mulA]#6
[h3] = ADD(mulA , mulB)<-[mulA]#6
[f0] = ADD(g0(2) , ss1(2))<-[ss1(2)]#7
[f7] = SUB(g0(2) , ss1(2))<-[ss1(2)]#7
[f1] = ADD(g1(2) , h3)<-[h3]#7
[f6] = SUB(g1(2) , h3)<-[h3]#7
[f2] = ADD(h1(2) , g3)<-[g3]#7
[f5] = SUB(h1(2) , g3)<-[g3]#7
```

```
  [f3] = ADD(h0(2) , qq1(2))<-[qq1(2)]#7
  [f4] = SUB(h0(2) , qq1(2))<-[qq1(2)]#7
  [sel10]=AND(hor_line_adres(6),3)<-[hor_line_adres(6)]#7
  [sel3] =SHR_AND(hor_line_adres(7),2,1)<-[hor_line_adres(7)]#8
  [reg1_wd_m1]=MULTIPLEX(sel10(2),f0(2),f1(3),f2(4),f3(5))<-[sel10(2)]
  [reg1_wd_m2]=MULTIPLEX(sel10(2),f4(6),f5(7),f6(8),f7(9))<-[sel10(2)]
  [reg1_wd]=MULTIPLEX(sel3(2),reg1_wd_m1,reg1_wd_m2,0,0)<-[sel3(2)]#9
  [reg2_wd_m1]=MULTIPLEX(sel10(3),f0(2),f1(3),f2(4),f3(5))<-[sel10(3)]
  [reg2_wd_m2]=MULTIPLEX(sel10(3),f4(6),f5(7),f6(8),f7(9))<-[sel10(3)]
  [reg2_wd]=MULTIPLEX(sel3(3),reg2_wd_m1,reg2_wd_m2,0,0)<-[sel3(3)]
  [reg3_wd_m1]=MULTIPLEX(sel10(4),f0(2),f1(3),f2(4),f3(5))<-[sel10(4)]
  [reg3_wd_m2]=MULTIPLEX(sel10(4),f4(6),f5(7),f6(8),f7(9))<-[sel10(4)]
  [reg3_wd]=MULTIPLEX(sel3(4),reg3_wd_m1,reg3_wd_m2,0,0)<-[sel3(4)]
  [reg4_wd_m1]=MULTIPLEX(sel10(5),f0(2),f1(3),f2(4),f3(5))<-[sel10(5)]
  [reg4_wd_m2]=MULTIPLEX(sel10(5),f4(6),f5(7),f6(8),f7(9))<-[sel10(5)]
  [reg4_wd]=MULTIPLEX(sel3(5),reg4_wd_m1,reg4_wd_m2,0,0)<-[sel3(5)]
  [reg5_wd_m1]=MULTIPLEX(sel10(6),f0(2),f1(3),f2(4),f3(5))<-[sel10(6)]
  [reg5_wd_m2]=MULTIPLEX(sel10(6),f4(6),f5(7),f6(8),f7(9))<-[sel10(6)]
  [reg5_wd]=MULTIPLEX(sel3(6),reg5_wd_m1,reg5_wd_m2,0,0)<-[sel3(6)]
  [reg6_wd_m1]=MULTIPLEX(sel10(7),f0(2),f1(3),f2(4),f3(5))<-[sel10(7)]
  [reg6_wd_m2]=MULTIPLEX(sel10(7),f4(6),f5(7),f6(8),f7(9))<-[sel10(7)]
  [reg6_wd]=MULTIPLEX(sel3(7),reg6_wd_m1,reg6_wd_m2,0,0)<-[sel3(7)]
  [reg7_wd_m1]=MULTIPLEX(sel10(8),f0(2),f1(3),f2(4),f3(5))<-[sel10(8)]
  [reg7_wd_m2]=MULTIPLEX(sel10(8),f4(6),f5(7),f6(8),f7(9))<-[sel10(8)]
  [reg7_wd]=MULTIPLEX(sel3(8),reg7_wd_m1,reg7_wd_m2,0,0)<-[sel3(8)]
  [reg8_wd_m1]=MULTIPLEX(sel10(9),f0(2),f1(3),f2(4),f3(5))<-[sel10(9)]
  [reg8_wd_m2]=MULTIPLEX(sel10(9),f4(6),f5(7),f6(8),f7(9))<-[sel10(9)]
  [reg8_wd]=MULTIPLEX(sel3(9),reg8_wd_m1,reg8_wd_m2,0,0)<-[sel3(9)]

  [reg1_read]=REG(ver_line_cnt(2),0,hor_line_adres(9) ,reg1_wd)<-[]
  [reg2_read]=REG(ver_line_cnt(2),0,hor_line_adres(10),reg2_wd)<-[]
  [reg3_read]=REG(ver_line_cnt(2),0,hor_line_adres(11),reg3_wd)<-[]
  [reg4_read]=REG(ver_line_cnt(2),0,hor_line_adres(12),reg4_wd)<-[]
  [reg5_read]=REG(ver_line_cnt(2),0,hor_line_adres(13),reg5_wd)<-[]
  [reg6_read]=REG(ver_line_cnt(2),0,hor_line_adres(14),reg6_wd)<-[]
  [reg7_read]=REG(ver_line_cnt(2),0,hor_line_adres(15),reg7_wd)<-[]
  [reg8_read]=REG(ver_line_cnt(2),0,hor_line_adres(16),reg8_wd)<-[]

[ver_line_cnt,ver_line_Exit]=SFOR_SMALLER(0,8,1,0)<-[hor_line_Exit(9)]
#[16384 4520 8867 12873  16384 19266  21407  22725]
# C_set1=[C4 C7 C6 C5 C4 C3 C2 C1];
 [ver_mul_1_0] = MUL_SHIFT(reg1_read,16384,15)<-[reg1_read]#2
 [ver_mul_1_1] = MUL_SHIFT(reg2_read,4520,15) <-[reg2_read]
 [ver_mul_1_2] = MUL_SHIFT(reg3_read,8867,15) <-[reg3_read]
 [ver_mul_1_3] = MUL_SHIFT(reg4_read,12873,15)<-[reg4_read]
 [ver_mul_1_4] = MUL_SHIFT(reg5_read,16384,15)<-[reg5_read]
 [ver_mul_1_5] = MUL_SHIFT(reg6_read,19266,15)<-[reg6_read]
 [ver_mul_1_6] = MUL_SHIFT(reg7_read,21407,15)<-[reg7_read]
 [ver_mul_1_7] = MUL_SHIFT(reg8_read,22725,15)<-[reg8_read]
#[16384 22725 21407  19266  16384  12873  8867   4520]
```

```
#C_set2=[C4 C1 C2 C3 C4 C5 C6 C7];
 [ver_mul_2_0]=MUL_SHIFT(reg1_read,16384,15)<-[reg1_read]
 [ver_mul_2_1]=MUL_SHIFT(reg2_read,22725,15)<-[reg2_read]
 [ver_mul_2_2]=MUL_SHIFT(reg3_read,21407,15)<-[reg3_read]
 [ver_mul_2_3]=MUL_SHIFT(reg4_read,19266,15)<-[reg4_read]
 [ver_mul_2_4]=MUL_SHIFT(reg5_read,16384,15)<-[reg5_read]
 [ver_mul_2_5]=MUL_SHIFT(reg6_read,12873,15)<-[reg6_read]
 [ver_mul_2_6]=MUL_SHIFT(reg7_read,8867,15)<-[reg7_read]
 [ver_mul_2_7]=MUL_SHIFT(reg8_read,4520,15)<-[reg8_read]
 [ver_Q1] = SUB(ver_mul_1_1, ver_mul_1_7)<-[ver_mul_1_7]
 [ver_Q0] = SUB(ver_mul_1_5, ver_mul_1_3)<-[ver_mul_1_5]
 [ver_S0] = ADD(ver_mul_2_5, ver_mul_2_3)<-[ver_mul_1_5]
 [ver_S1] = ADD(ver_mul_2_1, ver_mul_2_7)<-[ver_mul_2_7]
 [ver_p0] = ADD(ver_mul_1_0 , ver_mul_1_4)<-[ver_mul_1_4]
 [ver_p1] = SUB(ver_mul_2_0 , ver_mul_2_4)<-[ver_mul_2_4]
 [ver_r1] = SUB(ver_mul_1_2 , ver_mul_1_6)<-[ver_mul_1_6]
 [ver_r0] = ADD(ver_mul_2_2 , ver_mul_2_6)<-[ver_mul_2_6]

 [ver_ss1] = ADD(ver_S1, ver_S0)<-[ver_S1]
 [ver_qq1] = ADD(ver_Q1, ver_Q0)<-[ver_Q1]
 [ver_ss0] = SUB(ver_S1, ver_S0)<-[ver_S1]
 [ver_qq0] = SUB(ver_Q1, ver_Q0)<-[ver_Q1]
 [ver_g0] = ADD(ver_p0, ver_r0)<-[ver_r0]
 [ver_g1] = ADD(ver_p1, ver_r1)<-[ver_r1]
 [ver_h0] = SUB(ver_p0, ver_r0)<-[ver_r0]
 [ver_h1] = SUB(ver_p1, ver_r1)<-[ver_r1]
 [ver_mulA]=MUL_SHIFT(ver_ss0,23170,15)<-[ver_ss0]
 [ver_mulB]=MUL_SHIFT(ver_qq0,23170,15)<-[ver_qq0]
 [ver_g3] = SUB(ver_mulA , ver_mulB)<-[ver_mulA]
 [ver_h3] = ADD(ver_mulA , ver_mulB)<-[ver_mulA]

 [ver_f0]=ADD(ver_g0(2) , ver_ss1(2)) <-[ver_ss1(2)]
 [ver_f7]=SUB(ver_g0(2) , ver_ss1(2)) <-[ver_ss1(2)]
 [ver_f1]=ADD(ver_g1(2) , ver_h3) <-[ver_h3]
 [ver_f6]=SUB(ver_g1(2) , ver_h3) <-[ver_h3]
 [ver_f2]=ADD(ver_h1(2) , ver_g3) <-[ver_g3]
 [ver_f5]=SUB(ver_h1(2) , ver_g3) <-[ver_g3]
 [ver_f3]=ADD(ver_h0(2) , ver_qq1(2))<-[ver_qq1(2)]
 [ver_f4]=SUB(ver_h0(2) , ver_qq1(2))<-[ver_qq1(2)]

 [ver_f0r1] = ADD(ver_f0,ver_f0)<-[ver_f0]
 [ver_f1r1] = ADD(ver_f1,ver_f1)<-[ver_f1]
 [ver_f2r1] = ADD(ver_f2,ver_f2)<-[ver_f2]
 [ver_f3r1] = ADD(ver_f3,ver_f3)<-[ver_f3]
 [ver_f4r1] = ADD(ver_f4,ver_f4)<-[ver_f4]
 [ver_f5r1] = ADD(ver_f5,ver_f5)<-[ver_f5]
 [ver_f6r1] = ADD(ver_f6,ver_f6)<-[ver_f6]
 [ver_f7r1] = ADD(ver_f7,ver_f7)<-[ver_f7]

 [ver_f0r2] = ADD(ver_f0r1,31)<-[ver_f0r1]
```

```
 [ver_f1r2] = ADD(ver_f1r1,31)<-[ver_f1r1]
 [ver_f2r2] = ADD(ver_f2r1,31)<-[ver_f2r1]
 [ver_f3r2] = ADD(ver_f3r1,31)<-[ver_f3r1]
 [ver_f4r2] = ADD(ver_f4r1,31)<-[ver_f4r1]
 [ver_f5r2] = ADD(ver_f5r1,31)<-[ver_f5r1]
 [ver_f6r2] = ADD(ver_f6r1,31)<-[ver_f6r1]
 [ver_f7r2] = ADD(ver_f7r1,31)<-[ver_f7r1]

 [ver_f0s]=SAT(ver_f0r2,-16384,16383)<-[ver_f0r2];
 [ver_f1s]=SAT(ver_f1r2,-16384,16383)<-[ver_f1r2];
 [ver_f2s]=SAT(ver_f2r2,-16384,16383)<-[ver_f2r2];
 [ver_f3s]=SAT(ver_f3r2,-16384,16383)<-[ver_f3r2];
 [ver_f4s]=SAT(ver_f4r2,-16384,16383)<-[ver_f4r2];
 [ver_f5s]=SAT(ver_f5r2,-16384,16383)<-[ver_f5r2];
 [ver_f6s]=SAT(ver_f6r2,-16384,16383)<-[ver_f6r2];
 [ver_f7s]=SAT(ver_f7r2,-16384,16383)<-[ver_f7r2];
```

C.2 Maxval Algorithm

```
######################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
######################################################################

#IO Connections
%PI:INPUT
%max_result  :OUTPUT
#PI is the Primary Input pulse to initiate execution
[LoopStart, 0 ]=DELAY(PI)<-[PI]
#FOR loop generates an index value, i, every clock cycle
[i, i_Exit] = SFOR_SMALLER( 0,16,1,0)<-[LoopStart]
 #8 data are read from the memory
 [d1]    = MEM(0,i,aData1.txt,0,0)<-[]
 [d2]    = MEM(0,i,aData2.txt,0,0)<-[]
 [d3]    = MEM(0,i,aData3.txt,0,0)<-[]
 [d4]    = MEM(0,i,aData4.txt,0,0)<-[]
 [d5]    = MEM(0,i,aData5.txt,0,0)<-[]
 [d6]    = MEM(0,i,aData6.txt,0,0)<-[]
 [d7]    = MEM(0,i,aData7.txt,0,0)<-[]
 [d8]    = MEM(0,i,aData8.txt,0,0)<-[]
 #Each clock cycle, the maximum value is updated
 [m1(-32768)] = MAX(m1,0,d1,0)<-[d1,LoopStart(1)]
 [m2(-32768)] = MAX(m2,0,d2,0)<-[d2,LoopStart(1)]
 [m3(-32768)] = MAX(m3,0,d3,0)<-[d3,LoopStart(1)]
 [m4(-32768)] = MAX(m4,0,d4,0)<-[d4,LoopStart(1)]
 [m5(-32768)] = MAX(m5,0,d5,0)<-[d5,LoopStart(1)]
 [m6(-32768)] = MAX(m6,0,d6,0)<-[d6,LoopStart(1)]
 [m7(-32768)] = MAX(m7,0,d7,0)<-[d7,LoopStart(1)]
```

```
 [m8(-32768)] = MAX(m8,0,d8,0)<-[d8,LoopStart(1)]
 #after the FOR loop finishes, the MAX tree below
 #finds the maximum value.
 [m1_2] = MAX(m1,0,m2,1)<-[i_Exit(1)]
 [m3_4] = MAX(m3,2,m4,3)<-[i_Exit(1)]
 [m5_6] = MAX(m5,4,m6,5)<-[i_Exit(1)]
 [m7_8] = MAX(m7,6,m8,7)<-[i_Exit(1)]
 [m1_4] = MAX(m1_2,0,m3_4,0)<-[m1_2]
 [m5_8] = MAX(m5_6,0,m7_8,0)<-[m5_6]
 [max_result] = MAX(m1_4,0,m5_8,0)<-[m1_4]
```

C.3 Dot Product Algorithm

```
#####################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
#####################################################################
#PI is the Primary Input pulse to initiate execution
[LoopStart]=DELAY(PI)<-[PI]
 [i,i_Exit] = SFOR_SMALLER( 0,32,1,0)<-[LoopStart]
 [a.1]    = MEM(0,i(4),aData1.txt,0,0)<-[]
 [a.2]    = MEM(0,i(4),aData2.txt,0,0)<-[]
 [a.3]    = MEM(0,i(4),aData3.txt,0,0)<-[]
 [a.4]    = MEM(0,i(4),aData4.txt,0,0)<-[]
 [a.5]    = MEM(0,i(4),aData5.txt,0,0)<-[]
 [a.6]    = MEM(0,i(4),aData6.txt,0,0)<-[]
 [a.7]    = MEM(0,i(4),aData7.txt,0,0)<-[]
 [a.8]    = MEM(0,i(4),aData8.txt,0,0)<-[]

 [b.1]    = MEM(0,i(4),bData1.txt,0,0)<-[]
 [b.2]    = MEM(0,i(4),bData2.txt,0,0)<-[]
 [b.3]    = MEM(0,i(4),bData3.txt,0,0)<-[]
 [b.4]    = MEM(0,i(4),bData4.txt,0,0)<-[]
 [b.5]    = MEM(0,i(4),bData5.txt,0,0)<-[]
 [b.6]    = MEM(0,i(4),bData6.txt,0,0)<-[]
 [b.7]    = MEM(0,i(4),bData7.txt,0,0)<-[]
 [b.8]    = MEM(0,i(4),bData8.txt,0,0)<-[]

 [c_lsb.1:8,c_msb.1:8]=MUL_SHIFT(a.1:8,b.1:8,0)<-[a.1:8];
 [d_lsb.1:8(0),d_c.1:8]=ADD(d_lsb.1:8,c_lsb.1:8)<-[c_lsb.1:8,LoopStart(3)];
 [d_msb.1:8(0),0]=ADDC(d_msb.1:8,c_msb.1:8(1),d_c.1:8)<-[c_lsb.1:8(1),LoopStart(4)];
 [lsb_sum1_1,carry1_1] = ADD(d_lsb.1,d_lsb.2)<-[i_Exit(3)]
 [lsb_sum1_2,carry1_2] = ADD(d_lsb.3,d_lsb.4)<-[i_Exit(3)]
 [lsb_sum1_3,carry1_3] = ADD(d_lsb.5,d_lsb.6)<-[i_Exit(3)]
 [lsb_sum1_4,carry1_4] = ADD(d_lsb.7,d_lsb.8)<-[i_Exit(3)]

 [msb_sum1_1] = ADDC(d_msb.1,d_msb.2,carry1_1)<-[carry1_1]
 [msb_sum1_2] = ADDC(d_msb.3,d_msb.4,carry1_2)<-[carry1_2]
```

```
 [msb_sum1_3] = ADDC(d_msb.5,d_msb.6,carry1_3)<-[carry1_3]
 [msb_sum1_4] = ADDC(d_msb.7,d_msb.8,carry1_4)<-[carry1_4]

 [lsb_sum2_1,carry2_1] = ADD(lsb_sum1_1,lsb_sum1_2)<-[lsb_sum1_1]
 [lsb_sum2_2,carry2_2] = ADD(lsb_sum1_3,lsb_sum1_4)<-[lsb_sum1_3]
 [msb_sum2_1] = ADDC(msb_sum1_1,msb_sum1_2,carry2_1)<-[carry2_1]
 [msb_sum2_2] = ADDC(msb_sum1_3,msb_sum1_4,carry2_2)<-[carry2_2]
 [lsb_sum3_1,carry3_1] = ADD(lsb_sum2_1,lsb_sum2_2)<-[lsb_sum2_1]
 [msb_sum3_1] = ADDC(msb_sum2_1,msb_sum2_2,carry3_1)<-[carry3_1]
```

C.4 Maxidx Algorithm

```
######################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
######################################################################
#IO Connections
%PI:INPUT
%w_addr:INPUT
%w_d1:INPUT
%w_d2:INPUT
%w_d3:INPUT
%w_d4:INPUT
%w_d5:INPUT
%w_d6:INPUT
%w_d7:INPUT
%w_d8:INPUT
%max_index:OUTPUT
#PI is the Primary Input pulse to initiate execution
[LoopStart, 0 ]=DELAY(PI)<-[PI]
[i, i_Exit] = SFOR_SMALLER( 0,32,1,0)<-[LoopStart]
 [a.1,0]    = MEM(0,i(1),aData1.txt,w_addr,w_d1)<-[]
 [a.2,0]    = MEM(0,i(1),aData2.txt,w_addr,w_d2)<-[]
 [a.3,0]    = MEM(0,i(1),aData3.txt,w_addr,w_d3)<-[]
 [a.4,0]    = MEM(0,i(1),aData4.txt,w_addr,w_d4)<-[]
 [a.5,0]    = MEM(0,i(1),aData5.txt,w_addr,w_d5)<-[]
 [a.6,0]    = MEM(0,i(1),aData6.txt,w_addr,w_d6)<-[]
 [a.7,0]    = MEM(0,i(1),aData7.txt,w_addr,w_d7)<-[]
 [a.8,0]    = MEM(0,i(1),aData8.txt,w_addr,w_d8)<-[]

 [m.1:8(-32768),index.1:8] = MAX(m.1:8,index.1:8,a.1:8,i(2))<-[a.1:8,LoopStart(2)]
 [m1_2,idx1_1] = MAX(m.1,0,m.2,1)<-[i_Exit(2)]
 [m3_4,idx1_2] = MAX(m.3,2,m.4,3)<-[i_Exit(2)]
 [m5_6,idx1_3] = MAX(m.5,4,m.6,5)<-[i_Exit(2)]
 [m7_8,idx1_4] = MAX(m.7,6,m.8,7)<-[i_Exit(2)]
 [m1_4,idx2_1] = MAX(m1_2,idx1_1,m3_4,idx1_2)<-[m1_2]
 [m5_8,idx2_2] = MAX(m5_6,idx1_3,m7_8,idx1_4)<-[m5_6]
 [max_result,max_index] = MAX(m1_4,idx2_1,m5_8,idx2_2)<-[m1_4]
```

C.5 32-Tap FIR Fiter

```
#PI is the Primary Input pulse to initiate execution
[LoopStart, 0 ]=DELAY(PI)<-[PI]
 [i, i_Exit] = SFOR_SMALLER( 0,1024,1,0)<-[LoopStart]
 [data,0]    = MEM(0,i,data.txt,0,0)<-[]
 #qf=[-23 -39 87 182 -348 -638  1257  4095
 #     4095 1257  -638  -348   182  87   -39 -23];
 #data is multiplied with the first filter coefficient
 [mul0]  = MUL_SHIFT(data    ,-23 ,11)<-[data]
 #one cycle delayed data is multiplied with
 #the second filter coefficient.
 [mul1]  = MUL_SHIFT(data(1) ,-39 ,11)<-[data(1)]
 [mul2]  = MUL_SHIFT(data(2) ,87  ,11)<-[data(2)]
 [mul3]  = MUL_SHIFT(data(3) ,182 ,11)<-[data(3)]
 [mul4]  = MUL_SHIFT(data(4) ,-348,11)<-[data(4)]
 [mul5]  = MUL_SHIFT(data(5) ,-638,11)<-[data(5)]
 [mul6]  = MUL_SHIFT(data(6) ,1257,11)<-[data(6)]
 [mul7]  = MUL_SHIFT(data(7) ,4095,11)<-[data(7)]
 [mul8]  = MUL_SHIFT(data(8) ,4095,11)<-[data(8)]
 [mul9]  = MUL_SHIFT(data(9) ,1257,11)<-[data(9)]
 [mul10] = MUL_SHIFT(data(10),-638,11)<-[data(10)]
 [mul11] = MUL_SHIFT(data(11),-348,11)<-[data(11)]
 [mul12] = MUL_SHIFT(data(12),182 ,11)<-[data(12)]
 [mul13] = MUL_SHIFT(data(13),87  ,11)<-[data(13)]
 [mul14] = MUL_SHIFT(data(14),-39 ,11)<-[data(14)]
 [mul15] = MUL_SHIFT(data(15),-23 ,11)<-[data(15)]
 #adder tree stage-1
 [add1_0] = ADD(mul0 ,mul1)<-[mul0]
 [add1_1] = ADD(mul2 ,mul3)<-[mul2]
 [add1_2] = ADD(mul4 ,mul5)<-[mul4]
 [add1_3] = ADD(mul6 ,mul7)<-[mul6]
 [add1_4] = ADD(mul8 ,mul9)<-[mul8]
 [add1_5] = ADD(mul10,mul11)<-[mul10]
 [add1_6] = ADD(mul12,mul13)<-[mul12]
 [add1_7] = ADD(mul14,mul15)<-[mul14]
 #adder tree stage-2
 [add2_0] = ADD(add1_0 ,add1_1)<-[add1_0]
 [add2_1] = ADD(add1_2 ,add1_3)<-[add1_2]
 [add2_2] = ADD(add1_4 ,add1_5)<-[add1_4]
 [add2_3] = ADD(add1_6 ,add1_7)<-[add1_6]
 #adder tree stage-3
 [add3_0] = ADD(add2_0 ,add2_1)<-[add2_0]
 [add3_1] = ADD(add2_2 ,add2_3)<-[add2_2]
 #filter output
 [filter_out]   = ADD(add3_0 ,add3_1)<-[add3_0];
```

C.6 Vecsum Algorithm

```
#####################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
#####################################################################
#IO Connections
%PI:INPUT
%r_addr:INPUT
%d_1:OUTPUT
%d_2:OUTPUT
%d_3:OUTPUT
%d_4:OUTPUT
%d_5:OUTPUT
%d_6:OUTPUT
%d_7:OUTPUT
%d_8:OUTPUT
[LoopStart, 0 ]=DELAY(PI)<-[PI]
[i, i_Exit] = SFOR_SMALLER( 0,32,1,0)<-[LoopStart]
 [a.1]    = MEM(0,i(1),aData1.txt,0,0)<-[]
 [a.2]    = MEM(0,i(1),aData2.txt,0,0)<-[]
 [a.3]    = MEM(0,i(1),aData3.txt,0,0)<-[]
 [a.4]    = MEM(0,i(1),aData4.txt,0,0)<-[]
 [a.5]    = MEM(0,i(1),aData5.txt,0,0)<-[]
 [a.6]    = MEM(0,i(1),aData6.txt,0,0)<-[]
 [a.7]    = MEM(0,i(1),aData7.txt,0,0)<-[]
 [a.8]    = MEM(0,i(1),aData8.txt,0,0)<-[]
 [b.1]    = MEM(0,i(1),bData1.txt,0,0)<-[]
 [b.2]    = MEM(0,i(1),bData2.txt,0,0)<-[]
 [b.3]    = MEM(0,i(1),bData3.txt,0,0)<-[]
 [b.4]    = MEM(0,i(1),bData4.txt,0,0)<-[]
 [b.5]    = MEM(0,i(1),bData5.txt,0,0)<-[]
 [b.6]    = MEM(0,i(1),bData6.txt,0,0)<-[]
 [b.7]    = MEM(0,i(1),bData7.txt,0,0)<-[]
 [b.8]    = MEM(0,i(1),bData8.txt,0,0)<-[]

 [c.1:8]    = ADD(a.1:8,b.1:8)<-[a.1:8];
 #Above instruction is a vector addition
 [d.1:8]    = MEM(0,r_addr,0,i(3),c.1:8)<-[]
```

C.7 Fircplx Algoritm

```
#####################################################################
# C code reference of the algorithm can be found in
# http://focus.ti.com/en/download/dsp/c64plusbmarksasmfiles.zip
#####################################################################
[LoopStart, 0 ]=DELAY(PI)<-[PI]
```

```
[i,i_Exit] = SFOR_SMALLER( 0,256,1,0)<-[LoopStart]
[data_r]    = MEM(0,i,data_r.txt,0,0)<-[]
[data_i]    = MEM(0,i,data_i.txt,0,0)<-[]

[mul_rr.1] = MUL_SHIFT(data_r    ,-43   ,17)<-[data_r]
[mul_rr.2] = MUL_SHIFT(data_r(1) ,50    ,17)<-[data_r(1)]
[mul_rr.3] = MUL_SHIFT(data_r(2) ,186   ,17)<-[data_r(2)]
[mul_rr.4] = MUL_SHIFT(data_r(3) ,345   ,17)<-[data_r(3)]
[mul_rr.5] = MUL_SHIFT(data_r(4) ,417   ,17)<-[data_r(4)]
[mul_rr.6] = MUL_SHIFT(data_r(5) ,232   ,17)<-[data_r(5)]
[mul_rr.7] = MUL_SHIFT(data_r(6) ,-329  ,17)<-[data_r(6)]
[mul_rr.8] = MUL_SHIFT(data_r(7) ,-1190 ,17)<-[data_r(7)]
[mul_rr.9] = MUL_SHIFT(data_r(8) ,-1996 ,17)<-[data_r(8)]
[mul_rr.10] = MUL_SHIFT(data_r(9) ,-2168 ,17)<-[data_r(9)]
[mul_rr.11] = MUL_SHIFT(data_r(10),-1109 ,17)<-[data_r(10)]
[mul_rr.12] = MUL_SHIFT(data_r(11),1499  ,17)<-[data_r(11)]
[mul_rr.13] = MUL_SHIFT(data_r(12),5460  ,17)<-[data_r(12)]
[mul_rr.14] = MUL_SHIFT(data_r(13),10014 ,17)<-[data_r(13)]
[mul_rr.15] = MUL_SHIFT(data_r(14),14029 ,17)<-[data_r(14)]
[mul_rr.16] = MUL_SHIFT(data_r(15),16384 ,17)<-[data_r(15)]
[mul_rr.17] = MUL_SHIFT(data_r(16),16384 ,17)<-[data_r(16)]
[mul_rr.18] = MUL_SHIFT(data_r(17) ,14029,17)<-[data_r(17)]
[mul_rr.19] = MUL_SHIFT(data_r(18) ,10014,17)<-[data_r(18)]
[mul_rr.20] = MUL_SHIFT(data_r(19) ,5460 ,17)<-[data_r(19)]
[mul_rr.21] = MUL_SHIFT(data_r(20) ,1499 ,17)<-[data_r(20)]
[mul_rr.22] = MUL_SHIFT(data_r(21) ,-1109,17)<-[data_r(21)]
[mul_rr.23] = MUL_SHIFT(data_r(22) ,-2168,17)<-[data_r(22)]
[mul_rr.24] = MUL_SHIFT(data_r(23) ,-1996,17)<-[data_r(23)]
[mul_rr.25] = MUL_SHIFT(data_r(24) ,-1190,17)<-[data_r(24)]
[mul_rr.26] = MUL_SHIFT(data_r(25) ,-329 ,17)<-[data_r(25)]
[mul_rr.27] = MUL_SHIFT(data_r(26),232   ,17)<-[data_r(26)]
[mul_rr.28] = MUL_SHIFT(data_r(27),417   ,17)<-[data_r(27)]
[mul_rr.29] = MUL_SHIFT(data_r(28),345   ,17)<-[data_r(28)]
[mul_rr.30] = MUL_SHIFT(data_r(29),186   ,17)<-[data_r(29)]
[mul_rr.31] = MUL_SHIFT(data_r(30),50    ,17)<-[data_r(30)]
[mul_rr.32] = MUL_SHIFT(data_r(31),-43   ,17)<-[data_r(31)]

[mul_ri.1]  = MUL_SHIFT(data_r    ,-43   ,17)<-[data_r]
[mul_ri.2]  = MUL_SHIFT(data_r(1) ,50    ,17)<-[data_r(1)]
[mul_ri.3]  = MUL_SHIFT(data_r(2) ,186   ,17)<-[data_r(2)]
[mul_ri.4]  = MUL_SHIFT(data_r(3) ,345   ,17)<-[data_r(3)]
[mul_ri.5]  = MUL_SHIFT(data_r(4) ,417   ,17)<-[data_r(4)]
[mul_ri.6]  = MUL_SHIFT(data_r(5) ,232   ,17)<-[data_r(5)]
[mul_ri.7]  = MUL_SHIFT(data_r(6) ,-329  ,17)<-[data_r(6)]
[mul_ri.8]  = MUL_SHIFT(data_r(7) ,-1190 ,17)<-[data_r(7)]
[mul_ri.9]  = MUL_SHIFT(data_r(8) ,-1996 ,17)<-[data_r(8)]
[mul_ri.10]  = MUL_SHIFT(data_r(9) ,-2168 ,17)<-[data_r(9)]
[mul_ri.11] = MUL_SHIFT(data_r(10),-1109 ,17)<-[data_r(10)]
[mul_ri.12] = MUL_SHIFT(data_r(11),1499  ,17)<-[data_r(11)]
[mul_ri.13] = MUL_SHIFT(data_r(12),5460  ,17)<-[data_r(12)]
```

```
[mul_ri.14] = MUL_SHIFT(data_r(13),10014 ,17)<-[data_r(13)]
[mul_ri.15] = MUL_SHIFT(data_r(14),14029 ,17)<-[data_r(14)]
[mul_ri.16] = MUL_SHIFT(data_r(15),16384 ,17)<-[data_r(15)]
[mul_ri.17] = MUL_SHIFT(data_r(16),16384 ,17)<-[data_r(16)]
[mul_ri.18] = MUL_SHIFT(data_r(17) ,14029,17)<-[data_r(17)]
[mul_ri.19] = MUL_SHIFT(data_r(18) ,10014,17)<-[data_r(18)]
[mul_ri.20] = MUL_SHIFT(data_r(19) ,5460 ,17)<-[data_r(19)]
[mul_ri.21] = MUL_SHIFT(data_r(20) ,1499 ,17)<-[data_r(20)]
[mul_ri.22] = MUL_SHIFT(data_r(21) ,-1109,17)<-[data_r(21)]
[mul_ri.23] = MUL_SHIFT(data_r(22) ,-2168,17)<-[data_r(22)]
[mul_ri.24] = MUL_SHIFT(data_r(23) ,-1996,17)<-[data_r(23)]
[mul_ri.25] = MUL_SHIFT(data_r(24) ,-1190,17)<-[data_r(24)]
[mul_ri.26] = MUL_SHIFT(data_r(25) ,-329 ,17)<-[data_r(25)]
[mul_ri.27] = MUL_SHIFT(data_r(26),232   ,17)<-[data_r(26)]
[mul_ri.28] = MUL_SHIFT(data_r(27),417   ,17)<-[data_r(27)]
[mul_ri.29] = MUL_SHIFT(data_r(28),345   ,17)<-[data_r(28)]
[mul_ri.30] = MUL_SHIFT(data_r(29),186   ,17)<-[data_r(29)]
[mul_ri.31] = MUL_SHIFT(data_r(30),50    ,17)<-[data_r(30)]
[mul_ri.32] = MUL_SHIFT(data_r(31),-43   ,17)<-[data_r(31)]

[mul_ii.1]  = MUL_SHIFT(data_i    ,-43   ,17)<-[data_i]
[mul_ii.2]  = MUL_SHIFT(data_i(1) ,50    ,17)<-[data_i(1)]
[mul_ii.3]  = MUL_SHIFT(data_i(2) ,186   ,17)<-[data_i(2)]
[mul_ii.4]  = MUL_SHIFT(data_i(3) ,345   ,17)<-[data_i(3)]
[mul_ii.5]  = MUL_SHIFT(data_i(4) ,417   ,17)<-[data_i(4)]
[mul_ii.6]  = MUL_SHIFT(data_i(5) ,232   ,17)<-[data_i(5)]
[mul_ii.7]  = MUL_SHIFT(data_i(6) ,-329  ,17)<-[data_i(6)]
[mul_ii.8]  = MUL_SHIFT(data_i(7) ,-1190 ,17)<-[data_i(7)]
[mul_ii.9]  = MUL_SHIFT(data_i(8) ,-1996 ,17)<-[data_i(8)]
[mul_ii.10] = MUL_SHIFT(data_i(9) ,-2168 ,17)<-[data_i(9)]
[mul_ii.11] = MUL_SHIFT(data_i(10),-1109 ,17)<-[data_i(10)]
[mul_ii.12] = MUL_SHIFT(data_i(11),1499  ,17)<-[data_i(11)]
[mul_ii.13] = MUL_SHIFT(data_i(12),5460  ,17)<-[data_i(12)]
[mul_ii.14] = MUL_SHIFT(data_i(13),10014 ,17)<-[data_i(13)]
[mul_ii.15] = MUL_SHIFT(data_i(14),14029 ,17)<-[data_i(14)]
[mul_ii.16] = MUL_SHIFT(data_i(15),16384 ,17)<-[data_i(15)]
[mul_ii.17] = MUL_SHIFT(data_i(16),16384 ,17)<-[data_i(16)]
[mul_ii.18] = MUL_SHIFT(data_i(17) ,14029,17)<-[data_i(17)]
[mul_ii.19] = MUL_SHIFT(data_i(18) ,10014,17)<-[data_i(18)]
[mul_ii.20] = MUL_SHIFT(data_i(19) ,5460 ,17)<-[data_i(19)]
[mul_ii.21] = MUL_SHIFT(data_i(20) ,1499 ,17)<-[data_i(20)]
[mul_ii.22] = MUL_SHIFT(data_i(21) ,-1109,17)<-[data_i(21)]
[mul_ii.23] = MUL_SHIFT(data_i(22) ,-2168,17)<-[data_i(22)]
[mul_ii.24] = MUL_SHIFT(data_i(23) ,-1996,17)<-[data_i(23)]
[mul_ii.25] = MUL_SHIFT(data_i(24) ,-1190,17)<-[data_i(24)]
[mul_ii.26] = MUL_SHIFT(data_i(25) ,-329 ,17)<-[data_i(25)]
[mul_ii.27] = MUL_SHIFT(data_i(26),232   ,17)<-[data_i(26)]
[mul_ii.28] = MUL_SHIFT(data_i(27),417   ,17)<-[data_i(27)]
[mul_ii.29] = MUL_SHIFT(data_i(28),345   ,17)<-[data_i(28)]
[mul_ii.30] = MUL_SHIFT(data_i(29),186   ,17)<-[data_i(29)]
```

```
[mul_ii.31] = MUL_SHIFT(data_i(30),50    ,17)<-[data_i(30)]
[mul_ii.32] = MUL_SHIFT(data_i(31),-43   ,17)<-[data_i(31)]

[mul_ir.1]  = MUL_SHIFT(data_i    ,-43   ,17)<-[data_i]
[mul_ir.2]  = MUL_SHIFT(data_i(1) ,50    ,17)<-[data_i(1)]
[mul_ir.3]  = MUL_SHIFT(data_i(2) ,186   ,17)<-[data_i(2)]
[mul_ir.4]  = MUL_SHIFT(data_i(3) ,345   ,17)<-[data_i(3)]
[mul_ir.5]  = MUL_SHIFT(data_i(4) ,417   ,17)<-[data_i(4)]
[mul_ir.6]  = MUL_SHIFT(data_i(5) ,232   ,17)<-[data_i(5)]
[mul_ir.7]  = MUL_SHIFT(data_i(6) ,-329  ,17)<-[data_i(6)]
[mul_ir.8]  = MUL_SHIFT(data_i(7) ,-1190 ,17)<-[data_i(7)]
[mul_ir.9]   = MUL_SHIFT(data_i(8) ,-1996 ,17)<-[data_i(8)]
[mul_ir.10] = MUL_SHIFT(data_i(9) ,-2168 ,17)<-[data_i(9)]
[mul_ir.11] = MUL_SHIFT(data_i(10),-1109 ,17)<-[data_i(10)]
[mul_ir.12] = MUL_SHIFT(data_i(11),1499  ,17)<-[data_i(11)]
[mul_ir.13] = MUL_SHIFT(data_i(12),5460  ,17)<-[data_i(12)]
[mul_ir.14] = MUL_SHIFT(data_i(13),10014 ,17)<-[data_i(13)]
[mul_ir.15] = MUL_SHIFT(data_i(14),14029 ,17)<-[data_i(14)]
[mul_ir.16] = MUL_SHIFT(data_i(15),16384 ,17)<-[data_i(15)]
[mul_ir.17] = MUL_SHIFT(data_i(16),16384 ,17)<-[data_i(16)]
[mul_ir.18] = MUL_SHIFT(data_i(17) ,14029,17)<-[data_i(17)]
[mul_ir.19] = MUL_SHIFT(data_i(18) ,10014,17)<-[data_i(18)]
[mul_ir.20] = MUL_SHIFT(data_i(19) ,5460 ,17)<-[data_i(19)]
[mul_ir.21] = MUL_SHIFT(data_i(20) ,1499 ,17)<-[data_i(20)]
[mul_ir.22] = MUL_SHIFT(data_i(21) ,-1109,17)<-[data_i(21)]
[mul_ir.23] = MUL_SHIFT(data_i(22) ,-2168,17)<-[data_i(22)]
[mul_ir.24] = MUL_SHIFT(data_i(23) ,-1996,17)<-[data_i(23)]
[mul_ir.25] = MUL_SHIFT(data_i(24) ,-1190,17)<-[data_i(24)]
[mul_ir.26] = MUL_SHIFT(data_i(25) ,-329 ,17)<-[data_i(25)]
[mul_ir.27] = MUL_SHIFT(data_i(26),232   ,17)<-[data_i(26)]
[mul_ir.28] = MUL_SHIFT(data_i(27),417   ,17)<-[data_i(27)]
[mul_ir.29] = MUL_SHIFT(data_i(28),345   ,17)<-[data_i(28)]
[mul_ir.30] = MUL_SHIFT(data_i(29),186   ,17)<-[data_i(29)]
[mul_ir.31] = MUL_SHIFT(data_i(30),50    ,17)<-[data_i(30)]
[mul_ir.32] = MUL_SHIFT(data_i(31),-43   ,17)<-[data_i(31)]
#LRC has support for vector operations.
[real.1:32]  = SUB(mul_rr.1:32 ,mul_ii.1:32)<-[mul_rr.1:32]
[imag.1:32]  = ADD(mul_ri.1:32 ,mul_ir.1:32)<-[mul_ri.1:32]
#Adder Tree Stage-1:real parts
[add1_0_r]  = ADD(real.1 ,real.2)  <-[real.1]
[add1_1_r]  = ADD(real.3 ,real.4)  <-[real.3]
[add1_2_r]  = ADD(real.5 ,real.6)  <-[real.5]
[add1_3_r]  = ADD(real.7 ,real.8)  <-[real.7]
[add1_4_r]  = ADD(real.9 ,real.10) <-[real.9]
[add1_5_r]  = ADD(real.11,real.12) <-[real.11]
[add1_6_r]  = ADD(real.13,real.14) <-[real.13]
[add1_7_r]  = ADD(real.15,real.16) <-[real.15]
[add1_8_r]  = ADD(real.17 ,real.18)<-[real.17]
[add1_9_r]  = ADD(real.19 ,real.20)<-[real.19]
[add1_10_r] = ADD(real.21 ,real.22)<-[real.21]
```

```
[add1_11_r] = ADD(real.23 ,real.24)<-[real.23]
[add1_12_r] = ADD(real.25 ,real.26)<-[real.25]
[add1_13_r] = ADD(real.27, real.28)<-[real.27]
[add1_14_r] = ADD(real.29, real.30)<-[real.29]
[add1_15_r] = ADD(real.31, real.32)<-[real.31]
#Adder Tree Stage-1:imaginary parts
[add1_0_i]  = ADD(imag.1  ,imag.2)  <-[imag.1]
[add1_1_i]  = ADD(imag.3  ,imag.4)  <-[imag.3]
[add1_2_i]  = ADD(imag.5  ,imag.6)  <-[imag.5]
[add1_3_i]  = ADD(imag.7  ,imag.8)  <-[imag.7]
[add1_4_i]  = ADD(imag.9  ,imag.10) <-[imag.9]
[add1_5_i]  = ADD(imag.11 ,imag.12) <-[imag.11]
[add1_6_i]  = ADD(imag.13 ,imag.14) <-[imag.13]
[add1_7_i]  = ADD(imag.15 ,imag.16) <-[imag.15]
[add1_8_i]  = ADD(imag.17 ,imag.18) <-[imag.17]
[add1_9_i]  = ADD(imag.19 ,imag.20) <-[imag.19]
[add1_10_i] = ADD(imag.21 ,imag.22) <-[imag.21]
[add1_11_i] = ADD(imag.23 ,imag.24) <-[imag.23]
[add1_12_i] = ADD(imag.25 ,imag.26) <-[imag.25]
[add1_13_i] = ADD(imag.27, imag.28) <-[imag.27]
[add1_14_i] = ADD(imag.29, imag.30) <-[imag.29]
[add1_15_i] = ADD(imag.31, imag.32) <-[imag.31]
#Adder Tree Stage-2:real parts
[add2_0_r] = ADD(add1_0_r  ,add1_1_r) <-[add1_0_r]
[add2_1_r] = ADD(add1_2_r  ,add1_3_r) <-[add1_2_r]
[add2_2_r] = ADD(add1_4_r  ,add1_5_r) <-[add1_4_r]
[add2_3_r] = ADD(add1_6_r  ,add1_7_r) <-[add1_6_r]
[add2_4_r] = ADD(add1_8_r  ,add1_9_r) <-[add1_8_r]
[add2_5_r] = ADD(add1_10_r ,add1_11_r)<-[add1_10_r]
[add2_6_r] = ADD(add1_12_r ,add1_13_r)<-[add1_12_r]
[add2_7_r] = ADD(add1_14_r ,add1_15_r)<-[add1_14_r]
#Adder Tree Stage-2:imaginary parts
[add2_0_i] = ADD(add1_0_i  ,add1_1_i) <-[add1_0_i]
[add2_1_i] = ADD(add1_2_i  ,add1_3_i) <-[add1_2_i]
[add2_2_i] = ADD(add1_4_i  ,add1_5_i) <-[add1_4_i]
[add2_3_i] = ADD(add1_6_i  ,add1_7_i) <-[add1_6_i]
[add2_4_i] = ADD(add1_8_i  ,add1_9_i) <-[add1_8_i]
[add2_5_i] = ADD(add1_10_i ,add1_11_i)<-[add1_10_i]
[add2_6_i] = ADD(add1_12_i ,add1_13_i)<-[add1_12_i]
[add2_7_i] = ADD(add1_14_i ,add1_15_i)<-[add1_14_i]
#Adder Tree Stage-3:real parts
[add3_0_r] = ADD(add2_0_r ,add2_1_r)<-[add2_0_r]
[add3_1_r] = ADD(add2_2_r ,add2_3_r)<-[add2_2_r]
[add3_2_r] = ADD(add2_4_r ,add2_5_r)<-[add2_4_r]
[add3_3_r] = ADD(add2_6_r ,add2_7_r)<-[add2_6_r]
#Adder Tree Stage-3:imaginary parts
[add3_0_i] = ADD(add2_0_i ,add2_1_i)<-[add2_0_i]
[add3_1_i] = ADD(add2_2_i ,add2_3_i)<-[add2_2_i]
[add3_2_i] = ADD(add2_4_i ,add2_5_i)<-[add2_4_i]
[add3_3_i] = ADD(add2_6_i ,add2_7_i)<-[add2_6_i]
```

```
 #Adder Tree Stage-4:real parts
 [add4_0_r]   = ADD(add3_0_r ,add3_1_r)<-[add3_0_r];
 [add4_1_r]   = ADD(add3_2_r ,add3_3_r)<-[add3_2_r];
 #Adder Tree Stage-4:imaginary parts
 [add4_0_i]   = ADD(add3_0_i ,add3_1_i)<-[add3_0_i];
 [add4_1_i]   = ADD(add3_2_i ,add3_3_i)<-[add3_2_i];
 #Real and Imaginary Filter Outputs
 [fout_r]   = ADD(add4_0_r ,add4_1_r)<-[add4_0_r];
 [fout_i]   = ADD(add4_0_i ,add4_1_i)<-[add4_0_i];
```

C.8 16-State Viterbi Algorithm

```
% PI:INPUT
% decoded:OUTPUT
[delay_out]=DELAY(PI)<-[PI]
[i, i_Exit ]=SFOR_SMALLER( 0,100,1,1)<-[delay_out]
 #Parity LLRs
 [Par1] = MEM(0,i,r_par1.txt,0,0)<-[]
 [Par2] = MEM(0,i,r_par2.txt,0,0)<-[]
 #Calculate gamma values
 [G1] = ADD_MM(Par2(1),Par1(1))<-[Par1(1)]
 [G2] = SUB   (Par2(1),Par1(1))<-[Par1(1)]
 [G3] = SUB   (Par1(1),Par2(1))<-[Par1(1)]
 [G4] = ADD   (Par1(1),Par2(1))<-[Par1(1)]
 #Calculate the branches for state-1
 [S1_metric1]=ADD(S1,G1(1))<-[G1(1)]
 [S1_metric2]=ADD(S2,G4(1))<-[G4(1)]
 #calculate max of the branches and surviving bit
 [S1(0),bit1]= MAX(S1_metric1,0,S1_metric2,1)<-[S1_metric1,delay_out(2)]

 [S2_metric1]=ADD(S3,G3(1))<-[G3(1)]
 [S2_metric2]=ADD(S4,G2(1))<-[G2(1)]
 [S2(-128),bit2]=MAX(S2_metric1,0,S2_metric2,1)<-[S2_metric1,delay_out(2)]

 [S3_metric1]=ADD(S5,G3(1))<-[G3(1)]
 [S3_metric2]=ADD(S6,G2(1))<-[G2(1)]
 [S3(-128),bit3]=MAX(S3_metric1,0,S3_metric2,1)<-[S3_metric1,delay_out(2)]

 [S4_metric1]=ADD(S7,G1(1))<-[G1(1)]
 [S4_metric2]=ADD(S8,G4(1))<-[G4(1)]
 [S4(-128),bit4]=MAX(S4_metric1,0,S4_metric2,1)<-[S4_metric1,delay_out(2)]

 [S5_metric1]=ADD(S9,G2(1))<-[G2(1)]
 [S5_metric2]=ADD(S10,G3(1))<-[G3(1)]
 [S5(-128),bit5]   = MAX(S5_metric1,0,S5_metric2,1)<-[S5_metric1,delay_out(2)]

 [S6_metric1]=ADD(S11,G4(1))<-[G4(1)]
 [S6_metric2]=ADD(S12,G1(1))<-[G1(1)]
```

```
[S6(-128),bit6]   = MAX(S6_metric1,0,S6_metric2,1)<-[S6_metric1,delay_out(2)]

[S7_metric1]=ADD(S13,G4(1))<-[G4(1)]
[S7_metric2]=ADD(S14,G1(1))<-[G1(1)]
[S7(-128),bit7]=MAX(S7_metric1,0,S7_metric2,1)<-[S7_metric1,delay_out(2)]

[S8_metric1]=ADD(S15,G2(1))<-[G2(1)]
[S8_metric2]=ADD(S16,G3(1))<-[G3(1)]
[S8(-128),bit8]=MAX(S8_metric1,0,S8_metric2,1)<-[S8_metric1,delay_out(2)]

[S9_metric1]=ADD(S1,G4(1))<-[G4(1)]
[S9_metric2]=ADD(S2,G1(1))<-[G1(1)]
[S9(-128),bit9]=MAX(S9_metric1,0,S9_metric2,1)<-[S9_metric1,delay_out(2)]

[S10_metric1]=ADD(S3,G2(1))<-[G2(1)]
[S10_metric2]=ADD(S4,G3(1))<-[G3(1)]
[S10(-128),bit10]=MAX(S10_metric1,0,S10_metric2,1)<-[S10_metric1,delay_out(2)]

[S11_metric1]=ADD(S5,G2(1))<-[G2(1)]
[S11_metric2]=ADD(S6,G3(1))<-[G3(1)]
[S11(-128),bit11]=MAX(S11_metric1,0,S11_metric2,1)<-[S11_metric1,delay_out(2)]

[S12_metric1]=ADD(S7,G4(1))<-[G4(1)]
[S12_metric2]=ADD(S8,G1(1))<-[G1(1)]
[S12(-128),bit12]=MAX(S12_metric1,0,S12_metric2,1)<-[S12_metric1,delay_out(2)]

[S13_metric1]=ADD(S9,G3(1))<-[G3(1)]
[S13_metric2]=ADD(S10,G2(1))<-[G2(1)]
[S13(-128),bit13]=MAX(S13_metric1,0,S13_metric2,1)<-[S13_metric1,delay_out(2)]

[S14_metric1]=ADD(S11,G1(1))<-[G1(1)]
[S14_metric2]=ADD(S12,G4(1))<-[G4(1)]
[S14(-128),bit14]=MAX(S14_metric1,0,S14_metric2,1)<-[S14_metric1,delay_out(2)]

[S15_metric1]=ADD(S13,G1(1))<-[G1(1)]
[S15_metric2]=ADD(S14,G4(1))<-[G4(1)]
[S15(-128),bit15]=MAX(S15_metric1,0,S15_metric2,1)<-[S15_metric1,delay_out(2)]

[S16_metric1]=ADD(S15,G3(1))<-[G3(1)]
[S16_metric2]=ADD(S16,G2(1))<-[G2(1)]
[S16(-128),bit16]=MAX(S16_metric1,0,S16_metric2,1)<-[S16_metric1,delay_out(2)]

[bit_vector_1_4]   = MERGE(0,bit1,bit2,bit3,bit4)<-[bit1]
[bit_vector_5_8]   = MERGE(0,bit5,bit6,bit7,bit8)<-[bit5]
[bit_vector_9_12]  = MERGE(0,bit9,bit10,bit11,bit12)<-[bit9]
[bit_vector_13_16] = MERGE(0,bit13,bit14,bit15,bit16)<-[bit13]

[bit_vector_1_16] = MERGE(2,bit_vector_1_4,bit_vector_5_8,
            bit_vector_9_12,bit_vector_13_16)<-[bit_vector_1_4]
```

```
 [read_bit_vector,0]    = MEM(0,k,0,i(8),bit_vector_1_16)<-[];
 [max_S1_S2,  index_S1_S2]     = MAX(S1, 0, S2, 1) <-[i_Exit(5)]
 [max_S3_S4,  index_S3_S4]     = MAX(S3, 2, S4, 3) <-[i_Exit(5)]
 [max_S5_S6,  index_S5_S6]     = MAX(S5, 4, S6, 5) <-[i_Exit(5)]
 [max_S7_S8,  index_S7_S8]     = MAX(S7, 6, S8, 7) <-[i_Exit(5)]
 [max_S9_S10, index_S9_S10]    = MAX(S9, 8, S10,9) <-[i_Exit(5)]
 [max_S11_S12,index_S11_S12]   = MAX(S11,10,S12,11)<-[i_Exit(5)]
 [max_S13_S14,index_S13_S14]   = MAX(S13,12,S14,13)<-[i_Exit(5)]
 [max_S15_S16,index_S15_S16]   = MAX(S15,14,S16,15)<-[i_Exit(5)]

 [max_S1_S4,index_S1_S4]=MAX(max_S1_S2,index_S1_S2,max_S3_S4,index_S3_S4)
                        <-[max_S1_S2]

 [max_S5_S8,index_S5_S8]=MAX(max_S5_S6, index_S5_S6,max_S7_S8,index_S7_S8)
                        <-[max_S5_S6]
 [max_S9_S12,index_S9_S12]=MAX(max_S9_S10,index_S9_S10,max_S11_S12,index_S11_S12)
                          <-[max_S9_S10]
 [max_S13_S16,index_S13_S16]=MAX(max_S13_S14, index_S13_S14,max_S15_S16,index_S15_S16)
                            <-[max_S13_S14]
 [max_S1_S8,index_S1_S8]=MAX(max_S1_S4, index_S1_S4, max_S5_S8,index_S5_S8)
                          <-[max_S1_S4]
 [max_S9_S16,index_S9_S16]=MAX(max_S9_S12, index_S9_S12,max_S13_S16,index_S13_S16)
                          <-[max_S9_S12]
 [max_State,index_State]=MAX(max_S1_S8,index_S1_S8, max_S9_S16,index_S9_S16)
                        <-[max_S1_S8]

[k, k_Exit ]=SFOR_BIGGER( 99,-1,-1,2)<-[ index_State ]
# back ward state computation is optimized for the
# generator polynomial
 [bw_bit_new]=SHR_AND(read_bit_vector,bw_state,1)<-[read_bit_vector]
 [bw_index,0]   = SHL_OR(bw_state,1,bw_bit_new)<-[bw_bit_new]
 [decoded,0]    = SHR_AND(bw_index,4,1)<-[bw_index];
 [bw_state(index_State),0]=AND(bw_index,15)<-[bw_index,index_State]
 #max_State inits bw_state
```

C.9 UMTS Turbo Decoder Algorithm

```
%PI:INPUT
%Le2:OUTPUT
[delay_out ]=DELAY(PI)<-[PI]
[k_m,0] = SELF_MUX(k,m)<-[]
[iter,iter_Exit]=FOR_SMALLER(0,10,1)<-[delay_out,SISO_Exit(1)]
  [SISO, SISO_Exit ] = FOR_SMALLER( 0,2,1)<-[iter,m_Exit(14)]
    [k, k_Exit ] = SFOR_SMALLER( 0,100,1,1)<-[SISO]
      [par1] = MEM(0,k_m,rx_par1.txt,0,0)<-[]
      [par2] = MEM(0,k_m,rx_par2.txt,0,0)<-[]
      [interleaver] = MEM(0,k_m,interleaver.txt,0,0)<-[]
      [0,inter_index] = EQUAL(SISO,0,k_m(2),interleaver(1))<-[k_m(2)]
```

```
  [syst  ] = MEM(0,inter_index,rx_sys.txt,0,0)<-[]
  [extr  ] = MEM(0,inter_index,0,wr_extr,Le2)<-[];
  [0,par ] = EQUAL(SISO,1,par2(2),par1(2))<-[par1(2)]
  [g1    ] = ADD(syst,extr)<-[syst]

  par_d1=par(1)#Renaming is possible in LRC
                  #here par(1), one cycle delayed version of par
                  #renamed as par_d1
  [g3 ] = ADD(g1,par_d1)<-[g1]
  #below are just renaming
  g0_aligned = 0
  g1_aligned = g1(1)
  g2_aligned = par(2)
  g3_aligned = g3

  s0_A = s0(1)
  [s0_B ] = ADD(s1,g3_aligned)<-[k(7)]
  [s1_A ] = ADD(s3,g2_aligned)<-[k(7)]
  [s1_B ] = ADD(s2,g1_aligned)<-[k(7)]
  [s2_A ] = ADD(s4,g2_aligned)<-[k(7)]
  [s2_B ] = ADD(s5,g1_aligned)<-[k(7)]
  [s3_A ] = ADD(s7,g0_aligned)<-[k(7)]
  [s3_B ] = ADD(s6,g3_aligned)<-[k(7)]
  [s4_A ] = ADD(s1,g0_aligned)<-[k(7)]
  [s4_B ] = ADD(s0,g3_aligned)<-[k(7)]
  [s5_A ] = ADD(s2,g2_aligned)<-[k(7)]
  [s5_B ] = ADD(s3,g1_aligned)<-[k(7)]
  [s6_A ] = ADD(s5,g2_aligned)<-[k(7)]
  [s6_B ] = ADD(s4,g1_aligned)<-[k(7)]
  [s7_A ] = ADD(s6,g0_aligned)<-[k(7)]
  [s7_B ] = ADD(s7,g3_aligned)<-[k(7)]

  [s0(0) ]     =MAX(s0_A,0,s0_B,1)<-[s0_B,SISO(4)];
  [s1(-16384) ]=MAX(s1_A,0,s1_B,1)<-[s1_A,SISO(4)];
  [s2(-16384) ]=MAX(s2_A,0,s2_B,1)<-[s2_A,SISO(4)];
  [s3(-16384) ]=MAX(s3_A,0,s3_B,1)<-[s3_A,SISO(4)];
  [s4(-16384) ]=MAX(s4_A,0,s4_B,1)<-[s4_A,SISO(4)];
  [s5(-16384) ]=MAX(s5_A,0,s5_B,1)<-[s5_A,SISO(4)];
  [s6(-16384) ]=MAX(s6_A,0,s6_B,1)<-[s6_A,SISO(4)];
  [s7(-16384) ]=MAX(s7_A,0,s7_B,1)<-[s7_A,SISO(4)];
  [wr_index(0) ]     = ADD(wr_index,1)<-[k(8),SISO]
  [alpha0] = MEM(0,m(6),alpha0.txt,wr_index,s0)<-[]
  [alpha1] = MEM(0,m(6),alpha1.txt,wr_index,s1)<-[]
  [alpha2] = MEM(0,m(6),alpha2.txt,wr_index,s2)<-[]
  [alpha3] = MEM(0,m(6),alpha3.txt,wr_index,s3)<-[]
  [alpha4] = MEM(0,m(6),alpha4.txt,wr_index,s4)<-[]
  [alpha5] = MEM(0,m(6),alpha5.txt,wr_index,s5)<-[]
  [alpha6] = MEM(0,m(6),alpha6.txt,wr_index,s6)<-[]
  [alpha7] = MEM(0,m(6),alpha7.txt,wr_index,s7)<-[]
[m, m_Exit ] = SFOR_BIGGER( 99,-1,-1,1)<-[k_Exit(11)]
```

```
 ss0_A = ss0(1)
[ss0_B ] = ADD(ss4,g3_aligned)<-[m(7)]
[ss1_A ] = ADD(ss4,g0_aligned)<-[m(7)]
[ss1_B ] = ADD(ss0,g3_aligned)<-[m(7)]
[ss2_A ] = ADD(ss5,g2_aligned)<-[m(7)]
[ss2_B ] = ADD(ss1,g1_aligned)<-[m(7)]
[ss3_A ] = ADD(ss1,g2_aligned)<-[m(7)]
[ss3_B ] = ADD(ss5,g1_aligned)<-[m(7)]
[ss4_A ] = ADD(ss2,g2_aligned)<-[m(7)]
[ss4_B ] = ADD(ss6,g1_aligned)<-[m(7)]
[ss5_A ] = ADD(ss6,g2_aligned)<-[m(7)]
[ss5_B ] = ADD(ss2,g1_aligned)<-[m(7)]
[ss6_A ] = ADD(ss7,g0_aligned)<-[m(7)]
[ss6_B ] = ADD(ss3,g3_aligned)<-[m(7)]
[ss7_A ] = ADD(ss3,g0_aligned)<-[m(7)]
[ss7_B ] = ADD(ss7,g3_aligned)<-[m(7)]
[ss0(0)] = MAX(ss0_A,0,ss0_B,1)<-[ss0_B,SISO(4)];
[ss1(0)] = MAX(ss1_A,0,ss1_B,1)<-[ss1_A,SISO(4)];
[ss2(0)] = MAX(ss2_A,0,ss2_B,1)<-[ss2_A,SISO(4)]
[ss3(0)] = MAX(ss3_A,0,ss3_B,1)<-[ss3_A,SISO(4)]
[ss4(0)] = MAX(ss4_A,0,ss4_B,1)<-[ss4_A,SISO(4)]
[ss5(0)] = MAX(ss5_A,0,ss5_B,1)<-[ss5_A,SISO(4)]
[ss6(0)] = MAX(ss6_A,0,ss6_B,1)<-[ss6_A,SISO(4)]
[ss7(0)] = MAX(ss7_A,0,ss7_B,1)<-[ss7_A,SISO(4)]

[metric0_0]=ADD(alpha0,ss0)<-[alpha0]
[metric0_1]=ADD(alpha1,ss0)<-[alpha1]
[metric1_0]=ADD(alpha3,ss1)<-[alpha3]
[metric1_1]=ADD(alpha2,ss1)<-[alpha2]
[metric2_0]=ADD(alpha4,ss2)<-[alpha4]
[metric2_1]=ADD(alpha5,ss2)<-[alpha5]
[metric3_0]=ADD(alpha7,ss3)<-[alpha7]
[metric3_1]=ADD(alpha6,ss3)<-[alpha6]
[metric4_0]=ADD(alpha1,ss4)<-[alpha1]
[metric4_1]=ADD(alpha0,ss4)<-[alpha0]
[metric5_0]=ADD(alpha2,ss5)<-[alpha2]
[metric5_1]=ADD(alpha3,ss5)<-[alpha3]
[metric6_0]=ADD(alpha5,ss6)<-[alpha5]
[metric6_1]=ADD(alpha4,ss6)<-[alpha4]
[metric7_0]=ADD(alpha6,ss7)<-[alpha6]
[metric7_1]=ADD(alpha7,ss7)<-[alpha7]
[gmetric0_0]=ADD(metric0_0,g0_aligned)<-[metric0_0]
[gmetric0_1]=ADD(metric0_1,g3_aligned)<-[metric0_1]
[gmetric1_0]=ADD(metric1_0,g2_aligned)<-[metric1_0]
[gmetric1_1]=ADD(metric1_1,g1_aligned)<-[metric1_1]
[gmetric2_0]=ADD(metric2_0,g2_aligned)<-[metric2_0]
[gmetric2_1]=ADD(metric2_1,g1_aligned)<-[metric2_1]
[gmetric3_0]=ADD(metric3_0,g0_aligned)<-[metric3_0]
[gmetric3_1]=ADD(metric3_1,g3_aligned)<-[metric3_1]
[gmetric4_0]=ADD(metric4_0,g0_aligned)<-[metric4_0]
```

```
  [gmetric4_1]=ADD(metric4_1,g3_aligned)<-[metric4_1]
  [gmetric5_0]=ADD(metric5_0,g2_aligned)<-[metric5_0]
  [gmetric5_1]=ADD(metric5_1,g1_aligned)<-[metric5_1]
  [gmetric6_0]=ADD(metric6_0,g2_aligned)<-[metric6_0]
  [gmetric6_1]=ADD(metric6_1,g1_aligned)<-[metric6_1]
  [gmetric7_0]=ADD(metric7_0,g0_aligned)<-[metric7_0]
  [gmetric7_1]=ADD(metric7_1,g3_aligned)<-[metric7_1]

  [max01_0]=MAX(gmetric0_0,0,gmetric1_0,1)<-[gmetric0_0]
  [max23_0]=MAX(gmetric2_0,0,gmetric3_0,1)<-[gmetric2_0]
  [max45_0]=MAX(gmetric4_0,0,gmetric5_0,1)<-[gmetric4_0]
  [max67_0]=MAX(gmetric6_0,0,gmetric7_0,1)<-[gmetric6_0]
  [max0123_0]=MAX(max01_0,0,max23_0,1)<-[max01_0]
  [max4567_0]=MAX(max45_0,0,max67_0,1)<-[max45_0]
  [max_all_0]=MAX(max0123_0,0,max4567_0,1)<-[max0123_0]
  [max01_1]=MAX(gmetric0_1,0,gmetric1_1,1)<-[gmetric0_1]
  [max23_1]=MAX(gmetric2_1,0,gmetric3_1,1)<-[gmetric2_1]
  [max45_1]=MAX(gmetric4_1,0,gmetric5_1,1)<-[gmetric4_1]
  [max67_1]=MAX(gmetric6_1,0,gmetric7_1,1)<-[gmetric6_1]

  [max0123_1]=MAX(max01_1,0,max23_1,1)<-[max01_1]
  [max4567_1]=MAX(max45_1,0,max67_1,1)<-[max45_1]
  [max_all_1]=MAX(max0123_1,0,max4567_1,1)<-[max0123_1]
  [Lall]=SUB(max_all_1,max_all_0)<-[max_all_1];
  [Le1]=SUB(Lall,syst(8))<-[Lall];
  [Le2]=SUB(Le1,extr(9))<-[Le1];

  [wr_extr]=DELAY(inter_index(10))<-[m(14)]
```

C.10 FFT Algorithm

```
#Radix-2 DIT 1024 Point FFT algorithm
#IO Connections
%PI:INPUT
%PI_ADDR:INPUT
%PI_DATA_REAL:INPUT
%PI_DATA_IMAG:INPUT
%DataReal:OUTPUT
%DataImag:OUTPUT

[delay_out, 0 ]=DELAY(PI)<-[PI]
#loop for stages:there are 10 stages in 1024 point FFT
[Stage,StageExit]=FOR_SMALLER(0,10,1)<-[delay_out(1),BflyExit(6)]
  [mask2(65535)] = SHL_OR(mask2,1,0)<-[Stage,delay_out(1)]
  [mask2_not] = NOT(mask2)<-[mask2]
  [mask1]  = SHR_OR(mask2_not,1,0)<-[mask2_not]
  [StageRev]  = SUB(9,Stage)<-[Stage]
  #loop  for butterflys
```

```
  [Bfly,  BflyExit  ] = SFOR_SMALLER( 0,512,1,1)<-[mask1]
    [TwiddleAdr1] = SHL_OR(Bfly(5),StageRev,0)<-[Bfly(5)]
    [TwiddleAdr] = AND(TwiddleAdr1,511)<-[TwiddleAdr1]
    [TwdReal]  = MEM(0,TwiddleAdr,TwiddleMemReal.txt,0,0)<-[]
    [TwdImag]  = MEM(0,TwiddleAdr,TwiddleMemImag.txt,0,0)<-[]

    [bflyadr2,  ]  = SHL_OR(Bfly,1,0)<-[Bfly]
    [bflymasked1]  = AND(Bfly,mask1)<-[Bfly]
    [bflymasked2]  = AND(bflyadr2,mask2)<-[bflyadr2]
    [bflyadr_a]    = OR(bflymasked1,bflymasked2)<-[bflymasked2]
    [bflyadr_b]    = OR(bflyadr_a,powerstage)<-[bflyadr_a]
    [DataAdres] = SELF_MUX(bflyadr_a,bflyadr_b)<-[]
    [AdresRev] = MEM(0,DataAdres,BitReverseMem.txt,0,0)<-[]

    [write_address]=SELF_MUX(AdresRev(7),PI_ADDR)<-[]
    [write_R]=SELF_MUX(W_Data_R,PI_DATA_REAL)<-[]
    [write_I]=SELF_MUX(W_Data_I,PI_DATA_IMAG)<-[]

    [DataReal]=MEM(0,AdresRev,DataMemReal.txt,write_address,write_R)<-[];
    [DataImag]=MEM(0,AdresRev,DataMemImag.txt,write_address,write_I)<-[];
    [oprA_Real] = DELAY(DataReal)<-[bflyadr_a(3)]
    [oprB_Real] = DELAY(DataReal)<-[bflyadr_b(3)]
    [oprA_Imag] = DELAY(DataImag)<-[bflyadr_a(3)]
    [oprB_Imag] = DELAY(DataImag)<-[bflyadr_b(3)]

    [RR] = MUL_SHIFT(oprB_Real,TwdReal,12)<-[oprB_Real]
    [RI] = MUL_SHIFT(oprB_Real,TwdImag,12)<-[oprB_Real]
    [IR] = MUL_SHIFT(oprB_Imag,TwdReal,12)<-[oprB_Real]
    [II] = MUL_SHIFT(oprB_Imag,TwdImag,12)<-[oprB_Real]

    [RealSum] = SUB(RR,II)<-[RR]
    [ImagSum] = ADD(RI,IR)<-[RI]

    [OutAReal] = ADD(oprA_Real(2),RealSum)<-[RealSum]
    [OutAImag] = ADD(oprA_Imag(2),ImagSum)<-[ImagSum]

    [OutBReal] = SUB(oprA_Real(2),RealSum)<-[RealSum]
    [OutBImag] = SUB(oprA_Imag(2),ImagSum)<-[ImagSum]

[W_Data_R] = SELF_MUX(OutAReal,OutBReal(1))<-[]
[W_Data_I] = SELF_MUX(OutAImag,OutBImag(1))<-[]

  [powerstage(1)] = SHL_OR(powerstage,1,0)
                  <-[BflyExit(5),delay_out(1)]
```

C.11 Multirate FIR Filter Algorithm

```
#IO Connections
```

```
%PI:INPUT
%w_addr:INPUT
%w_data:INPUT
%filter_out:OUTPUT

[LoopStart]=DELAY(PI)<-[PI]
[i, i_Exit] = SFOR_SMALLER( 0,1024,1,0)<-[LoopStart]
[data]    = MEM(0,i,data.txt,w_addr,w_data)<-[]
#qf=[-23 -39 87 182 -348 -638  1257  4095
#     4095 1257  -638  -348   182  87   -39 -23];
 [mul0]  = MUL_SHIFT(data    ,-23 ,11)<-[data]
 [mul1]  = MUL_SHIFT(data(2) ,-39 ,11)<-[data(2)]
 [mul2]  = MUL_SHIFT(data(4) ,87  ,11)<-[data(4)]
 [mul3]  = MUL_SHIFT(data(6) ,182 ,11)<-[data(6)]
 [mul4]  = MUL_SHIFT(data(8) ,-348,11)<-[data(8)]
 [mul5]  = MUL_SHIFT(data(10) ,-638,11)<-[data(10)]
 [mul6]  = MUL_SHIFT(data(12) ,1257,11)<-[data(12)]
 [mul7]  = MUL_SHIFT(data(14) ,4095,11)<-[data(14)]
 [mul8]  = MUL_SHIFT(data(16) ,4095,11)<-[data(16)]
 [mul9]  = MUL_SHIFT(data(18) ,1257,11)<-[data(18)]
 [mul10] = MUL_SHIFT(data(20),-638,11)<-[data(20)]
 [mul11] = MUL_SHIFT(data(22),-348,11)<-[data(22)]
 [mul12] = MUL_SHIFT(data(24),182 ,11)<-[data(24)]
 [mul13] = MUL_SHIFT(data(26),87  ,11)<-[data(26)]
 [mul14] = MUL_SHIFT(data(28),-39 ,11)<-[data(28)]
 [mul15] = MUL_SHIFT(data(30),-23 ,11)<-[data(30)]
 #adder tree stage-1
 [add1_0] = ADD(mul0 ,mul1)<-[mul0]
 [add1_1] = ADD(mul2 ,mul3)<-[mul2]
 [add1_2] = ADD(mul4 ,mul5)<-[mul4]
 [add1_3] = ADD(mul6 ,mul7)<-[mul6]
 [add1_4] = ADD(mul8 ,mul9)<-[mul8]
 [add1_5] = ADD(mul10,mul11)<-[mul10]
 [add1_6] = ADD(mul12,mul13)<-[mul12]
 [add1_7] = ADD(mul14,mul15)<-[mul14]
 #adder tree stage-2
 [add2_0] = ADD(add1_0 ,add1_1)<-[add1_0]
 [add2_1] = ADD(add1_2 ,add1_3)<-[add1_2]
 [add2_2] = ADD(add1_4 ,add1_5)<-[add1_4]
 [add2_3] = ADD(add1_6 ,add1_7)<-[add1_6]
 #adder tree stage-3
 [add3_0] = ADD(add2_0 ,add2_1)<-[add2_0]
 [add3_1] = ADD(add2_2 ,add2_3)<-[add2_2]
 [filter_out]   = ADD(add3_0 ,add3_1)<-[add3_0];
```

C.12 Multichannel FIR Filter

```
#IO Connections
```

```
%PI:INPUT
%filter_out:OUTPUT
[LoopStart, 0 ]=DELAY(PI)<-[PI]
[i, i_Exit] = SFOR_SMALLER( 0,1024,1,1)<-[LoopStart]
 [data_ch1]     = MEM(0,i,data.txt,0,0)<-[]
 [data_ch2]     = MEM(0,i(1),data1.txt,0,0)<-[]
 [data]         = SELF_MUX(data_ch1,data_ch2)<-[]
 #qf=[-23 -39 87 182 -348 -638  1257  4095
 #      4095 1257  -638  -348   182  87   -39 -23];
 [mul0]  = MUL_SHIFT(data    ,-23 ,11)<-[data]
 [mul1]  = MUL_SHIFT(data(2) ,-39 ,11)<-[data(2)]
 [mul2]  = MUL_SHIFT(data(4) ,87  ,11)<-[data(4)]
 [mul3]  = MUL_SHIFT(data(6) ,182 ,11)<-[data(6)]
 [mul4]  = MUL_SHIFT(data(8) ,-348,11)<-[data(8)]
 [mul5]  = MUL_SHIFT(data(10) ,-638,11)<-[data(10)]
 [mul6]  = MUL_SHIFT(data(12) ,1257,11)<-[data(12)]
 [mul7]  = MUL_SHIFT(data(14) ,4095,11)<-[data(14)]
 [mul8]  = MUL_SHIFT(data(16) ,4095,11)<-[data(16)]
 [mul9]  = MUL_SHIFT(data(18) ,1257,11)<-[data(18)]
 [mul10] = MUL_SHIFT(data(20),-638,11)<-[data(20)]
 [mul11] = MUL_SHIFT(data(22),-348,11)<-[data(22)]
 [mul12] = MUL_SHIFT(data(24),182 ,11)<-[data(24)]
 [mul13] = MUL_SHIFT(data(26),87  ,11)<-[data(26)]
 [mul14] = MUL_SHIFT(data(28),-39 ,11)<-[data(28)]
 [mul15] = MUL_SHIFT(data(30),-23 ,11)<-[data(30)]
 #adder tree stage-1
 [add1_0] = ADD(mul0 ,mul1)<-[mul0]
 [add1_1] = ADD(mul2 ,mul3)<-[mul2]
 [add1_2] = ADD(mul4 ,mul5)<-[mul4]
 [add1_3] = ADD(mul6 ,mul7)<-[mul6]
 [add1_4] = ADD(mul8 ,mul9)<-[mul8]
 [add1_5] = ADD(mul10,mul11)<-[mul10]
 [add1_6] = ADD(mul12,mul13)<-[mul12]
 [add1_7] = ADD(mul14,mul15)<-[mul14]
 #adder tree stage-2
 [add2_0] = ADD(add1_0 ,add1_1)<-[add1_0]
 [add2_1] = ADD(add1_2 ,add1_3)<-[add1_2]
 [add2_2] = ADD(add1_4 ,add1_5)<-[add1_4]
 [add2_3] = ADD(add1_6 ,add1_7)<-[add1_6]
 #adder tree stage-3
 [add3_0] = ADD(add2_0 ,add2_1)<-[add2_0]
 [add3_1] = ADD(add2_2 ,add2_3)<-[add2_2]
 [filter_out]   = ADD(add3_0 ,add3_1)<-[add3_0];
```