修士論文

FPGAへの命令・スケジューラの実装

Implementation of Instruction Scheduler on FPGA

指導教員　五島正裕 准教授

東京大学大学院 情報理工学系研究科

電子情報学専攻

48-096435

ジョーリ・アビシェク

Johri Abhishek

Abstract

This thesis describes an implementation technique of “Instruction Scheduler” on FPGA. This implementation of instruction scheduler is a part of the soft core version of an area efficient out-of-order superscalar microprocessor proposed by our lab.

Instruction scheduler is a very crucial part of out-of-order microprocessor, which decides the instructions to be issued for execution. It checks for the dependencies of instructions in the instruction window and issues a certain number of instructions when their dependencies are resolved and they are ready for execution. It is a critical component of microprocessor, which limits its operational clock speed.

To achieve high clock speed for the proposed FPGA implemented microprocessor, it is necessary that the instruction scheduler must achieve low data path delay on FPGA's special architecture.

Hence, in this report we propose an implementation technique for instruction scheduler on FPGA, taking into account the architecture specialties and restrictions of FPGA. The proposed implementation tries to reduce the critical path delay of instruction scheduler by making intelligent use of FPGA resources. We also propose an optimization technique for Select Logic.

Table of Contents

List of Figures

Chapter 1 Introduction

Field Programmable Gate Arrays (FPGA) have evolved a lot over the years, since they were first introduced in 1980s. Now, FPGAs are available on 40*nm* technology and have enormous amount of programmable logic. The modern FPGAs work on low voltages (~0.9V) consumes less power and come with inbuilt DSPs, RAM, Ethernet controller, Ultra-fast serial data transmission compatible transceivers etc feature. Thus, FPGAs are finding their way in many applications across variety of domains. The best feature of a FPGA comparing to an ASIC chip is that, FPGAs are reprogrammable with negligible cost. These features of FPGAs are also exploited by the soft core micro-processors. A soft core microprocessor can be easily customized to the specific needs of the application, by modifying individual components of the microprocessor.

In view of this, our lab is implementing an area efficient an *out-of-order* superscalar microprocessor on FPGA. This microprocessor was also proposed by our lab[2] and is under development.

This research is a part of the above mentioned FPGA implementation of the proposed microprocessor. As the architecture of FPGA is totally different than the ASIC, many techniques used in the proposed microprocessor, cannot be implemented directly on FPGA. And even if they are implemented on FPGA, they can further be optimized to make good use of FPGA architecture and resources.

In this research we propose a technique to implement "*Instruction Scheduler*" on FPGA. Instruction scheduler is a component of microprocessor which is responsible to perform *dynamic scheduling* for *out-of-order* microprocessors.

In an *out-of-order* microprocessor, instructions are executed out of the program order, but preserving the data flow and exception behavior. This helps to reduce un necessary frontend stalls and thus the processor throughput (IPC) is improved. The process to exploit this Instruction level Parallelism (ILP) by executing the instruction, out of program order, is called *Dynamic Scheduling*.

Instruction scheduler checks for the data dependencies of the instructions stored in instruction window and issues some instructions as soon as their dependencies are resolved and they get ready for execution. It is also partly responsible for taking care of structural hazards, by issuing instructions to the pipeline which are free and available to execute a new instruction.

Instruction scheduler is composed of two components, namely “Wakeup Logic” and “Select Logic”. These two components make a feedback loop and thus must not be pipelined in view of high *Instruction Per Cycle* (IPC) rate. Hence, instruction scheduler becomes a critical part of microprocessor, by becoming the slowest pipeline stage and limiting the overall operational clock speed of microprocessor.

To achieve high clock speed for the proposed FPGA implemented microprocessor, it is required that instruction scheduler achieve low critical path delay on FPGA's special architecture. In this report, we give an implementation technique for instruction scheduler on FPGA, which is optimized for the architecture specialties and restrictions of FPGA. The proposed implementation tries to reduce the critical path delay of instruction scheduler by making intelligent use of FPGA resources. We also propose an optimization technique for select logic. The proposed instruction scheduler consists of Matrix Scheduler for Wakeup logic and cyclic prefix-sum-thicket for Select Logic.

Organization of report : Chapter 2 covers the principle and concept of dynamic scheduling. It also covers the instruction scheduler, its two main components and their working. Chapter 3, introduces various techniques available for wakeup logic and select logic and their associated delays. Chapter 4 starts with the architecture of FPGA and give a brief introduction about the Xilinx Virtex-6 FPGA, which we are using for the implementation of instruction scheduler. It then explains the proposed implementation technique for the same. Chapter 5, provides with the implementation results and evaluation figures. Finally chapter 6, provides the conclusion for this research.

Chapter 2 Instruction Scheduler

This chapter explains about instruction scheduler and its structure. However, in order to understand the functioning of instruction scheduler, the concept of dynamic scheduling must be understood.

2.1 Dynamic Scheduling

Dynamic scheduling is an important technique to improve the *Instruction Per Cycle* (IPC) rate, especially in superscalar microprocessors. In this technique, hardware rearranges the instruction execution order to reduce some unnecessary stalls while maintaining the data flow and exception behavior[9]. Most of the modern out-of-order processors employ dynamic scheduling technique for improved performance.

The occurrence of unnecessary stall, and its elimination by rearranging the instructions can be understood with the help of following example:

```
...
MUL  R3, R1, R2
ADD  R4, R3, R5
DIV  R8, R6, R7
...
```

As there is a data dependence of `ADD` on `MUL`, it will stall the frontend of a statically scheduled pipeline till the result of `MUL` is available. And `DIV` must wait for the pipeline to resume, although it does have any dependence on previous instructions. Such hazards degrade the throughput of processor. However, it can be eliminated by simply changing the order of instruction execution.

This is accomplished by A) checking for any structural hazard while issuing and B) waiting till data hazard is eliminated and issue the instruction as soon as its operands are available. This results in *out-of-order* execution, and subsequently *out-of-order* completion.

Instruction scheduler is responsible for performing this dynamic scheduling. It has two main parts, wakeup logic and select logic. Section 2.1 and 2.2 explains about the wakeup logic and select logic respectively and in chapter 3, we discuss a few major techniques for both of these parts.

Before we discuss the functioning of wakeup and select logic, the concept of using tags in these logics must be understood.

Scheduling and tag

Conventional scheduling schemes uses a "tag" to uniquely address each instruction in the backend of the processor. These tags can be any unique identifiers. However, usually for the ease of writing back the result to the buffer, the designator of the buffer is used, because the tag value itself would be the designation. These tags are also used in wakeup operation to find out the matching consumer instruction.

During the rename phase, each instruction (consumer instruction: I_c) is assigned a tag according to the buffer entry designation. Let's call it tagD, as it refers to the destination (result) location of that instruction. In this phase I_c also finds the producer instruction's (I_p) tags for its left and right source operands[*]. tagD for I_p (L/R) will be referred as tagL/R.

During the wakeup phase, these tags are used to search for the consumers of the producer instruction's result. tagD of I_p is broadcasted and matched with tagL/R of each consumer instruction (I_c) in the window. If the tags matches, the corresponding ready bit rdyL/R is set representing that the result for that operand is now available.

2.2 Wakeup Logic

This section explains the functioning of the wakeup logic. Structures of conventional and other wakeup logics are explained in section 3.1

Wakeup logic is a part of issuing window, and is responsible for waking up the sleeping instructions. Sleeping instructions are those instructions which are waiting in the instruction window for their source operands to become available.

Consider the case of a series of instructions as shown in this example:

```
MUL R1, R8, R9        (1)
```

[*]For the simplicity of explanation, 2 source operands and 1 destination operand is assumed. However, the discussion can easily be extended for the case of more operands.

```
ADD R2, R1, #10       (2)
ADD R3, R1, #20       (3)
ADD R4, R1, #30       (4)
ADD R5, R3, R4        (5)
```

Initially, as 1st instruction does not depend on any instruction it will be issued. Next, we see that the left operand of the instruction number 2,3 and 4 depends on the result of 1st instruction, so they must not be issued for execution till the result of 1st instruction is available. Similarly, 5th instruction must not be issued till the result of 3rd and 4th instruction becomes available.

This process of waking up the sleeping the instructions to get them queued for the execution is controlled by the wakeup logic.

Lets review the above wakeup procedure in terms of tag processing. The ready bits rdyL/R of 1st instruction are already set, because it has no dependency. When this instruction is issued it becomes producer for the 2nd to 4th instructions. If it takes multiple cycles to get the result of 1st instruction, tagD of 1st instruction must not be broadcasted to wakeup logic till the data is available. Once the data of R1 is available (considering bypass), tagD of 1st instruction is broadcasted to wakeup logic and is matched with the tagL/R of rest of the instructions of window. Instructions 2 to 4 find that their tagL is the same as tagD and thus set their rdyL flag to 1. As their right operand does not have any dependency rdyR flag is already set. Now, as both the ready bits are set, instruction 2 to 4 are ready to be issued. Wakeup logic sets the request signal high for these instructions. These requests are then processed by the select logic to select the actual issuable instructions out of the given requests.

Tag matching for 5th instruction can be understood in similar fashion, with its producer instructions being instruction number 3 and 4.

2.3 Select Logic

Select logic is also a part of the issuing window along with the wakeup logic, it comes just next to wakeup logic. When the dependencies of any instruction are resolved during wakeup process, fundamentally it is ready for execution, but due to unavailability of execution units it may not be issued for execution.

Select logic selects a maximum of *Issue Width* (*IW*) number of instructions out of

Window Size (*WS*) or less number of “wokeup” requesting instructions. The maximum number of selected instructions depend on the available/free execution pipeline hardware recourses in the processor. These selected instructions are read from the issuing window and sent to execution unit. The output from select logic is called *grant*, as the requesting signals are granted permission for using execution units, by select logic.

Consider a case when the given integer superscalar processor is 2 way, i.e. has two integer execution pipelines. In such a case, from the example code in section 2.2, when the tagD of 1st instruction is broadcasted to wakeup logic, even though the dependencies of 2nd to 4th instructions are resolved simultaneously, but only 2 out of these 3 instructions can be issued in single cycle because of the limited availability of hardware execution pipelines.

Here comes the role of select logic, it selects *IW* (or less) number of instructions, depending on their priority. This priority is usually given to the oldest requesting instruction and then to the subsequent older instructions. Oldest instruction here refers to the instructions which occur earlier in the program order compared to other requesting instructions.

Thus, in the same example code, priority will be given to 2nd and 3rd instructions over the 4th instruction in a 2 way integer pipeline superscalar processor. In next cycle the rdyL/R bits of 4th instruction will still be 1, and its request will still be high, now select logic will select it for issuing (given the latency of ADD operation can be pipelined).

Selection Priority

A selection policy is used to decide which of the requesting signals get the grant. A study by Butler and Patt[5] showed that the overall performance is largely independent of the selection policy. In general, the selection priority as either the oldest to newest instruction in the window or from a particular location (usually, topmost) of instruction window. Select logic based on priority for a certain location (e.g. top), are easier to implement and has lesser delay as compared to the select logics based on priority of the oldest instruction. Latter has to take care of the head location (location of the oldest instruction), which can move back and forth during a program execution.

2.3.1 Cyclic Select Logic

If the reorder buffer is configured as wrap around buffer, it is not necessary that the request from oldest instruction, be at the top of all the requesting signals. Hence, a select logic with fix priority may select wrong requests to issue as newer instruction may occur above old instructions inside the window. There are two methods to deal with such situation.

1. **Shifting the requests**

   If the select logic gives priority to a fix position (usually the zeroth request), request signals must be logically shifted in wrap around fashion so that the location of the request signal by actual oldest instruction matches the location of the of the select logic's oldest instruction request input. Once the grant is provided by select logic, these grant signals must be shifted back the same amount in order to appropriately match with the window. Select logic along with the wakeup logic is already one of the highest delay components, and appending *WS* sized shifters before and after the select logic is not a good idea and practically not used.

2. **Accompanying head location in selection**

   In this method changes are made in select logic itself to comply with floating priority position. Usually each request is accompanied with a head signal. At any time only one of the *WS* head signals could be 1, representing location of the *head* (the oldest instruction in window). Such select logics are generally implemented by addition of requests to decide for grant. Addition is performed on request of each instruction starting from the *head* location and subsequently add the requests of each next instruction. Finally, grant is provided to a request signal at *n*th location (when location of head is taken as 0), if addition result of all the requests form head instruction to the instruction *n-1* (in *wrap around* fashion) is less than *IW*.

$$grant_n \ = \ req_n \ \& \ \left(\sum_{n-1}^{head} req \ < IW \right)$$

2.4 Pipelining

Wakeup and select logic together must perform their operation within one clock cycle. This is because the wakeup and the select logic make a feedback loop. The issued instructions from select phase are used as producer instructions I_p in wakeup phase. This creates a feedback from select to wakeup. If wakeup/select logic are pipelined over multiple pipe stages, dependent instructions cannot execute in consecutive cycles.

Consider if each process takes one clock cycle, as shown in Figure 2.1. The SUB instruction, whose left operand is dependent on ADD, cannot be issued in next clock cycle for execution, because the result of select phase would not be available to its wakeup phase and it keeps sleeping in window, till 3rd cycle. This pipeline bubble degrades the *IPC* by 15%, and is unlikely to meet the gain at clock speed [1]. Thus, wakeup and select logic together constitute an atomic operation and must be completed in single clock cycle for one cycle latency paths.

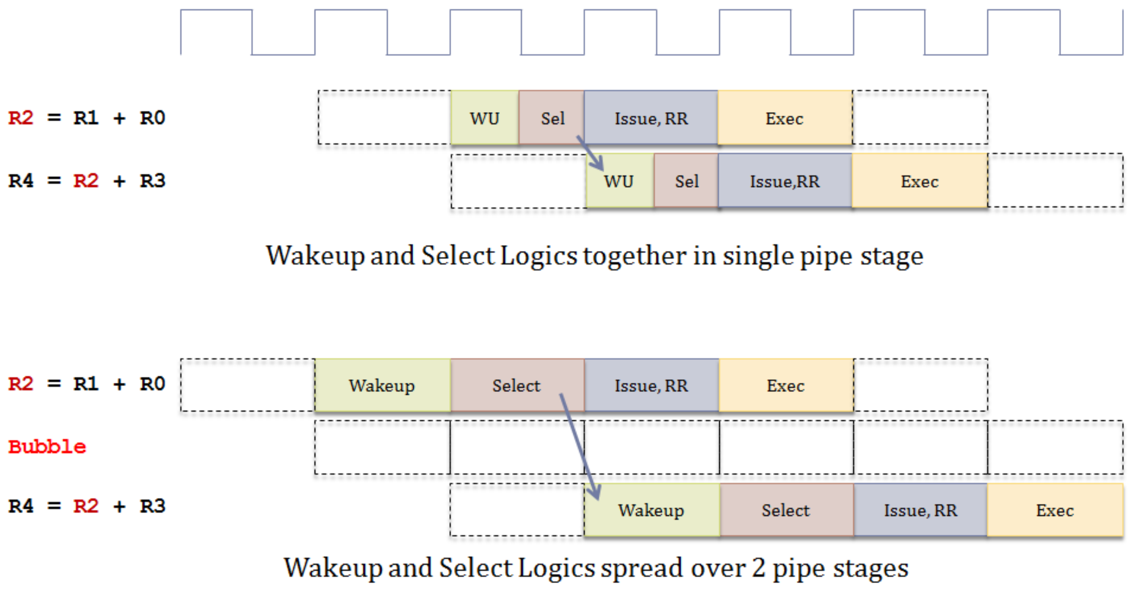

R2 = R1 + R0
R4 = R2 + R3
WU
Sel
Issue, RR
Exec
Issue,RR
Wakeup and Select Logics together in single pipe stage
Bubble
Wakeup
Select
Wakeup and Select Logics spread over 2 pipe stages

Figure 2.1 : Pipelining of Wakeup and Select logics

2.5 Decentralization

The issuing window can be divided in a set of smaller subwindows[3]. This decentralization considerably reduces the delay of instruction scheduler at the cost of small IPC penalties. Decentralization helps to reduce the effective size of each subwindow, and also enables latency optimization. This division typically takes place by making subwindows for Integer, floating point and load store type of instruction categories.

The structure of subwindows remains basically the same as of the original window, but the size is reduced by a factor q. Each subwindow can be thought of as a window with IW' and WS'. However, in the wakeup logic, each instruction in each subwindow must be matched with all the issued instructions, the inputs to wakeup logic cannot be reduced from IW to IW'.

Latency optimization can be achieved due to the fact that not all the instructions can be executed in one cycle, thus the paths with multi-cycle latency, can be pipelined for lower critical path delay.

Chapter 3 Various techniques of Instruction Scheduler

In this chapter we explain a few major techniques available to implement two main parts of instruction scheduler, namely wakeup and select logic. Section 3.1 explores the techniques about wakeup, while section 3.2 about select logic.

3.1 Wakeup Logic

This section explains 4 major techniques to implement wakeup logic. The traditional technique to implement wake up is to use CAM for matching the producer tag with the operand tags of consumer instruction. Second technique is to use multiport RAM instead of CAM. We also explain that this technique is practically not feasible. Another technique given by Henry et. al.[4] which uses *Cyclic Segmented Prefix* (CSP) circuits is also explained in this section. Finally, Matrix Scheduler technique is explained, which is used in the proposed microprocessor[2].

3.1.1 CAM based

The Wakeup logic conventionally uses *Content Addressable Memory* (CAM) to perform the waking up of sleeping instructions. Block diagram such wakeup logic is shown in Figure 3.1.

After the register renaming process, destination tag of the instruction is written in the memory assigned for destination tags. Let's call it ***tagD*-RAM.** This ***tagD*-RAM** is indexed by the grant signals and used for reading the tagD of instruction corresponding to the grant signal. Working of wakeup logic can be understood by the reading and writing of CAM.

Write: Source operands tags of the renamed consumer instructions are written in the CAM to their corresponding entry location's tagL and tagR fields.

Match & Read: Grant signals from select logic selects the corresponding destination tags form the ***tagD*-RAM**, and a maximum of *IW* destination tags are then broadcasted to the CAM, where each of these tags are compared with both the source operand tags (tagL and tagR) of all the entries in CAM. If the tags matches, corresponding ready bits (rdyL/R) are set. This is accomplished by logical OR*ing* the result for all the comparisons between that tagL/R and *IW* number of input tags.

Now, when both the ready bits of any instruction are set, the dependencies are thus resolved and that instruction is ready to be issued, hence it makes a request to select logic. This request signal is generated by taking a logical AND of rdyL/R.

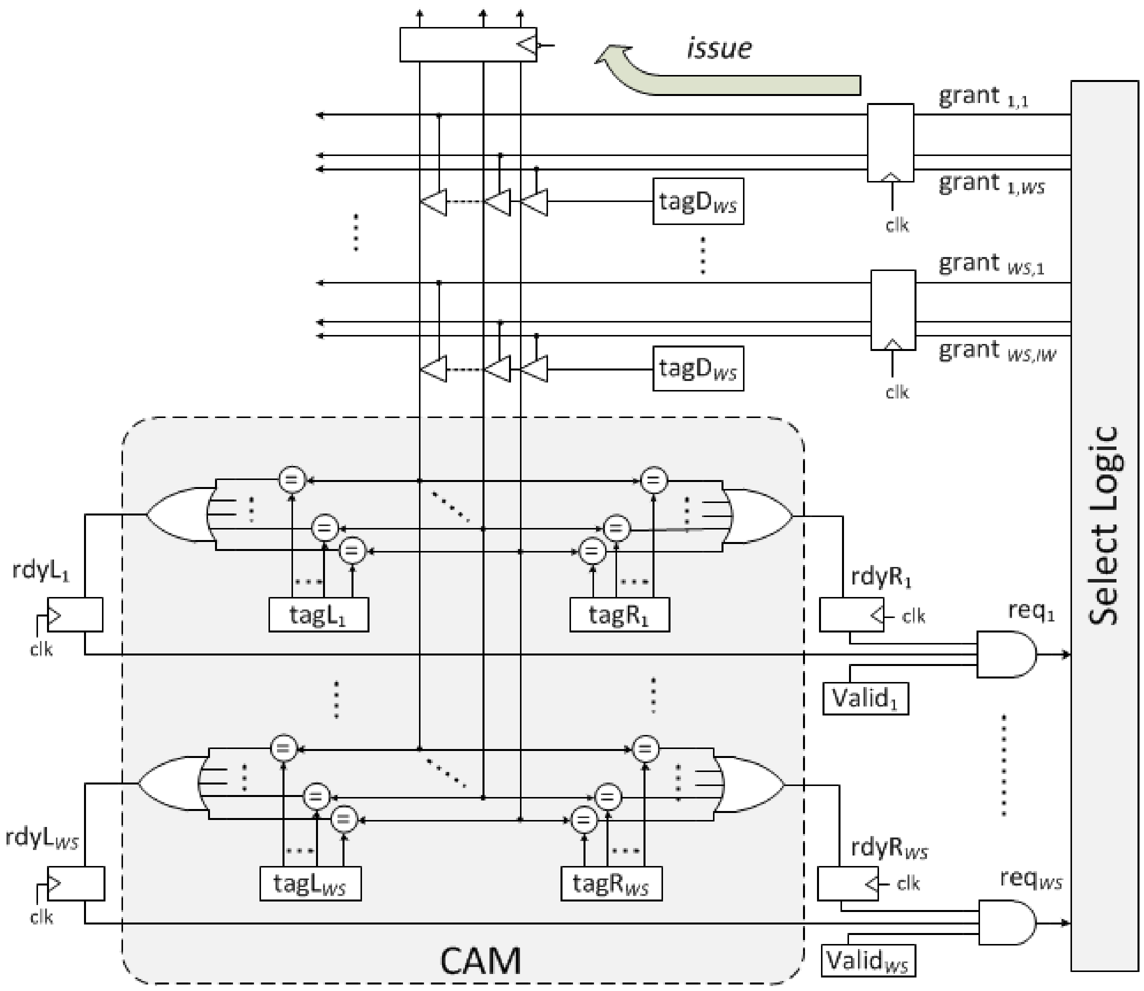


Figure 3.1 : CAM based Wakeup Logic

Total comparators required for this method are $IW \times WS \times$ *Num of Operands*. Each comparator is of the size of Tag width. Thus, this technique consumes lot more power and have high latency [6].

3.1.2 RAM based

Figure 3.2 shows a block diagram of a RAM based simple technique for wakeup logic. In this technique a 1 bit wide RAM is required with its number of words equal to the number of physical registers (*NR*). Each word of this RAM holds a ready bit corresponding to the physical resister number which is same as its word address. Thus, let's call it ***rdy* RAM**.

With inputs to this wakeup being the same as CAM based technique, explained in previous sub section, working of this RAM can be understood with the write and read of ***rdy*RAM**

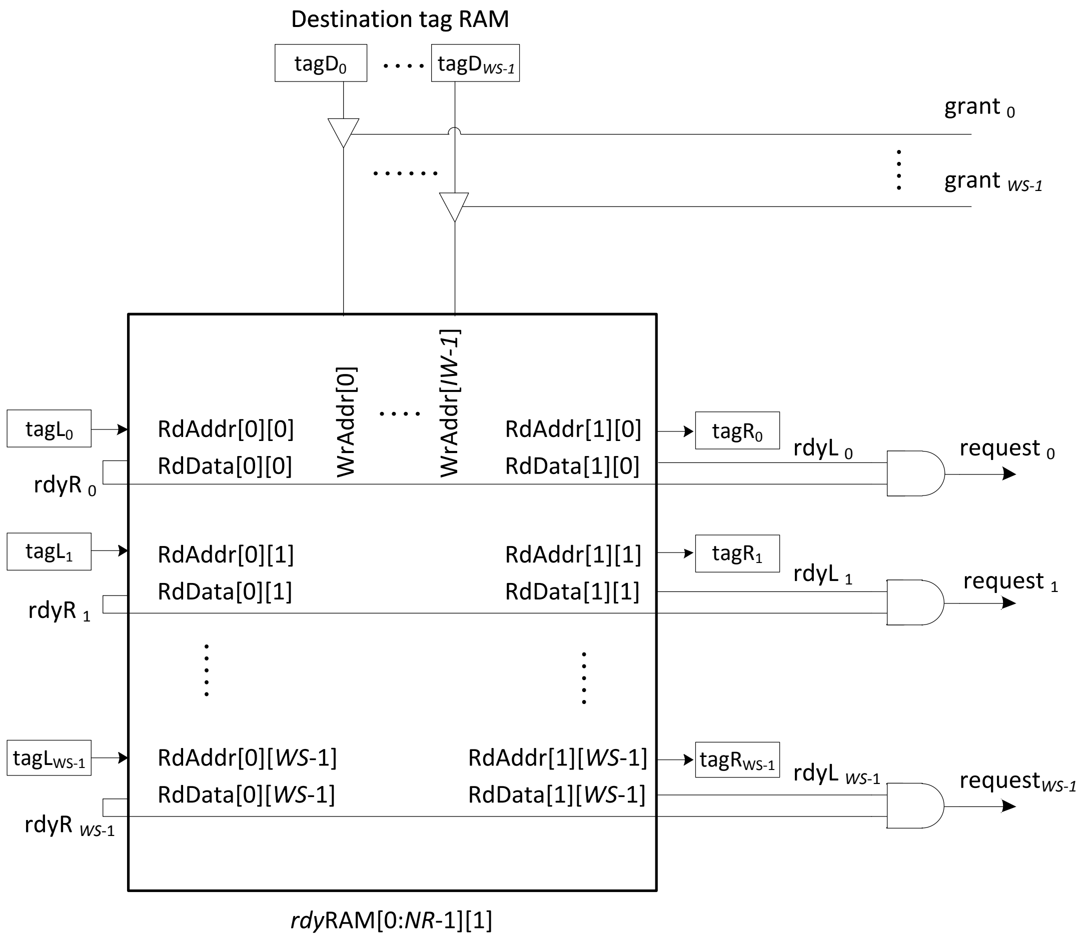


Figure 3.2 : Block diagram for RAM based wakeup logic

Write : Using grant signals as the index, the producer tags (tagD) are read from ***tagD*-RAM**, and passed on to the ***rdy*RAM**. Now, ***rdy*RAM** uses these values as the addresses and writes 1 at the asserted tagD entries, representing that the result of instructions corresponding to those tagD are now available.

Read: Consumer instructions in window, reads ***rdy*RAM** using their tagL/R tags as address lines. The values read, are the rdyL/R bits and can be used to generate request by taking their bitwise AND.

This method requires that the ***rdy*RAM** has *IW* number of independent write ports and 2×*WS* number of independent read ports. Taking an example of window size of 8, total read ports required would be 16 (2×8). In current technology, a RAM with such large number of read ports is practically not feasible.

3.1.3 CSP circuit based

A whole different type of implementation for wakeup logic was given by Henry et. al.[4], using *Cyclic Segmented Prefix* (CSP) circuits. This technique uses one CSP circuit for each logical register defined in the processor's instruction set architecture. The technique is equally valid for physical registers instead of logical registers, if register renaming is applied.

The dedicated CSP circuit for each register informs buffer, whether the register's value is available or not. Each CSP circuit operates independently of the others. Each instruction in the buffer uses multiplexers for each operand to select the ready bit rdyL/R of all the register's ready bits.

Each block inside the CSP circuit receives two inputs; one is the signal indicating that the value for corresponding register is available (i.e. *grant* from select logic), and the other signal is tells whether that instruction writes the corresponding register. *Head* instruction is an exception for these signals as it uses them to initialize the register as ready.

Structure of such wakeup logic implementation is shown in Figure 3.3(a). Considering 32 logical registers in processor. Each instruction in the reordering buffer provides its operand register info to all the CSP circuits and receives 32 ready bits indicating the readiness of each register. Each instruction uses 32-to-1 multiplexers (not shown in figure) to select its operand's ready bit.

Linear gate-delay implementation of the wake-up process for instructions depending on register R5 is shown in Figure 3.3(b). Figure shows the path for the availability of R5 in the CSP circuit. Bold lines express high value in the wire. Input bit for each block of CSP circuit is driven by the g*rant* form select logic. Each instruction in the reordering buffer sets its segment bit high if its result is stored in register R5. Again, form the Figure, the grant for instruction F has been awarded and it announces its availability of R5 by setting its input bit high. On the other hand another producer of R5, the instruction C is yet to be issued. Hence the ready bit for left operand of instruction G is set, while instruction E's ready bit will not be set, because it is suppose to use the value of R5 coming from instruction C which is not yet available.

The CSP circuit itself can be implemented with linear gate delay or with different types of logarithmic gate delays. Few logarithmic gate delay CSP circuits are explained in section 4.2 for select logic, however they can be applied in wakeup logic also only with a minor change of replacing the addition operator with bypass wire.

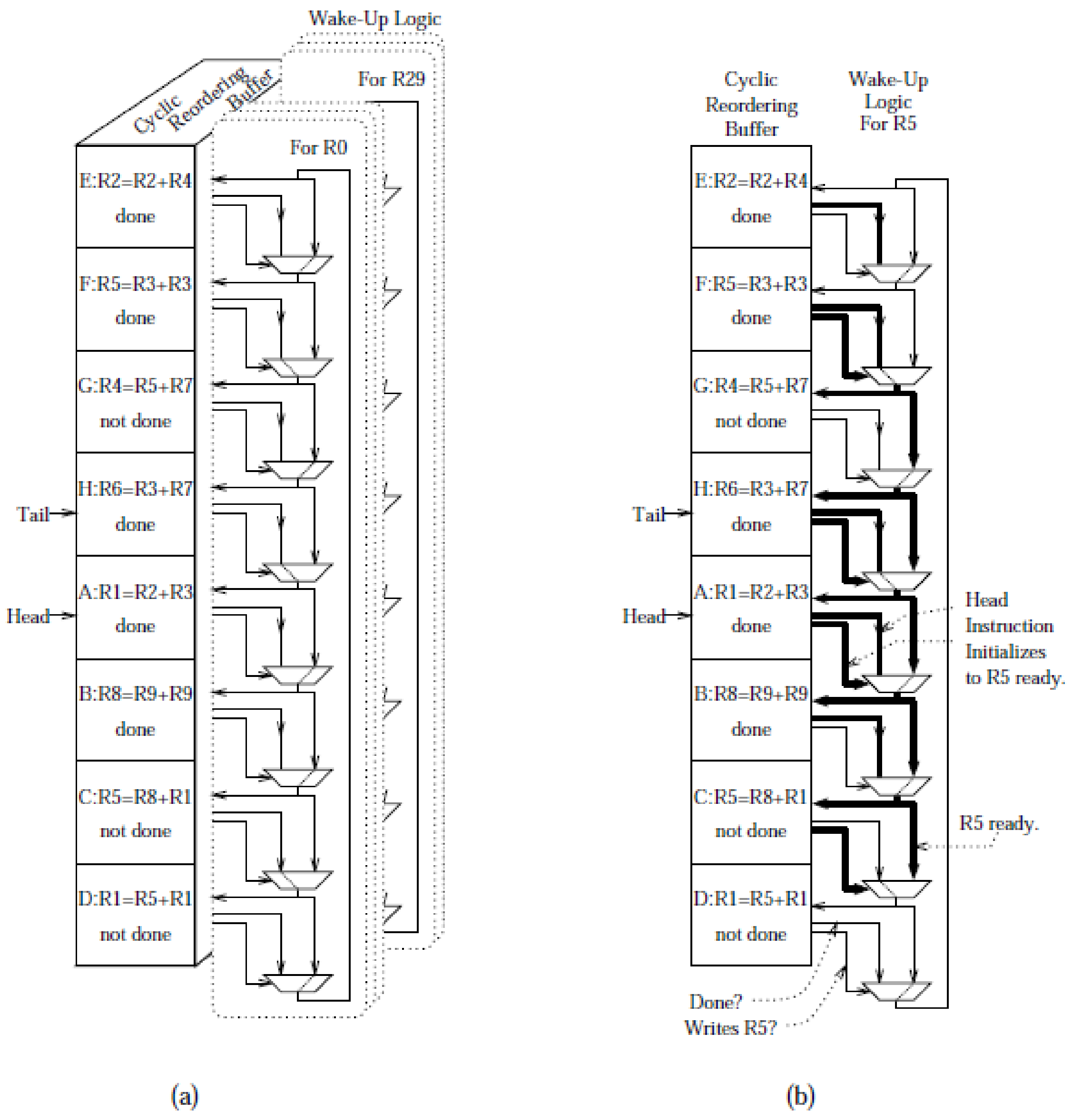


Figure 3.3† **: (a) CSP Circuit based wakeup logic, with 8-entry wrap-around reorder buffer and 32 logical registers. (b) Wakeup operation for instructions depending on register R5**

† Borrowed from [2]

3.1.4 Matrix Scheduler

Goshima et al[1] proposed a technique which eliminates the high cost CAM from the wakeup logic, by implementing RAM based dependency matrices. This technique reduces the complexity of wakeup logic by moving the heavy parts of the dependence detection required for the wakeup operation to the frontend of the processor. The associative search of the CAM scheme is replaced by simply reading the dependency matrices.

Matrix scheduler consists of two $WS \times WS$ dependence matrices, one for each of the left and right source operands. An example of such a matrix is as shown in Figure 3.4. The c-th row and the p-th column (c:consumer, p:producer) element of the left/right matrix is set to 1 if the left/right source operand of the c-th row is the same as the destination register of the p-th instruction. In other words, each row of the matrix is a bit vector which indicates the producer of the left/right source operand of the c-th instruction.

Consider the example of instructions shown in Figure 3.4. In the instruction window there are four instructions stored in continuous manner. First instruction does not depend on any other instruction so its corresponding row is all zero. The second and third instructions' left operand depends on the result of first instruction, thus the first bit of their left matrix's bit vectors are set to 1. Similarly, because the fourth instruction's left operand depends on the result of second instruction and right operand depends on the result of third instruction, the second bit of 4-th row in left matrix and the third bit of the 4-th row in right matrix are set to 1.

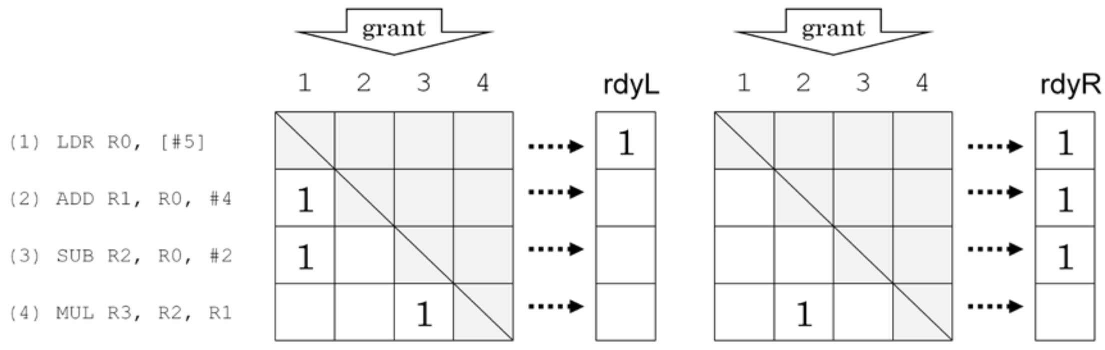


Figure 3.4 : Left and right dependency matrices for Matrix Scheduler

As stated earlier, the wakeup operation is performed by reading the matrices. When the instructions are issued, their corresponding *IW* columns in each matrix are bitwise ORed and the resulting column bit-vectors of each matrix shows the ready bits rdyL/R of instructions' left/right operand. Now simply ANDing the corresponding rdyL/R bits gives the request signal for each instruction in the window. In the example shown in Figure 3.4, when the first instruction is issued, only column 1 is read and as the consumer instructions 2 and 3 have 1 in their first column, the ready bit for their left operands becomes 1. As the ready bit for their right operand is already 1, ANDing the two ready bit column vectors sets the request signal active for the 2nd and 3rd instructions. In the next clock cycle (or a predefined number of cycles) if 2nd and 3rd instructions are issued simultaneously, the 3rd and 2nd column in each of the left and right matrices are bitwise ORed, setting the ready bits (rdyL/R) of 4th instruction to 1. Now, as both the operands are ready, their AND result will provide the request signal for the 4th instruction.

Block diagram of matrix scheduler is shown in figure 3.5. Grant signals from selection logic are used to read columns of matrix and set the rdyL/R bits. As it is visible from the figure, matrix scheduler does not require tag comparisons and as a result CAM and RAM portion of the conventional CAM based wakeup are replaced by dependency matrices. Although reading of tagD is still be required but for issue purpose only. For bitwise ORing of multiple column vectors a small modification in RAM structure is proposed in the scheme. With this modification, the bitwise OR operation can be performed along with the reading of RAM itself.

Matrix update

Producer tables are used to update the matrices. Producer tables are the same as the register map table but holds the mapping from the logical register numbers to the producer designators, instead of the tags. The dependence detector required for this logic is also the same as that of the rename logic and it can be shared between the rename and this logic.

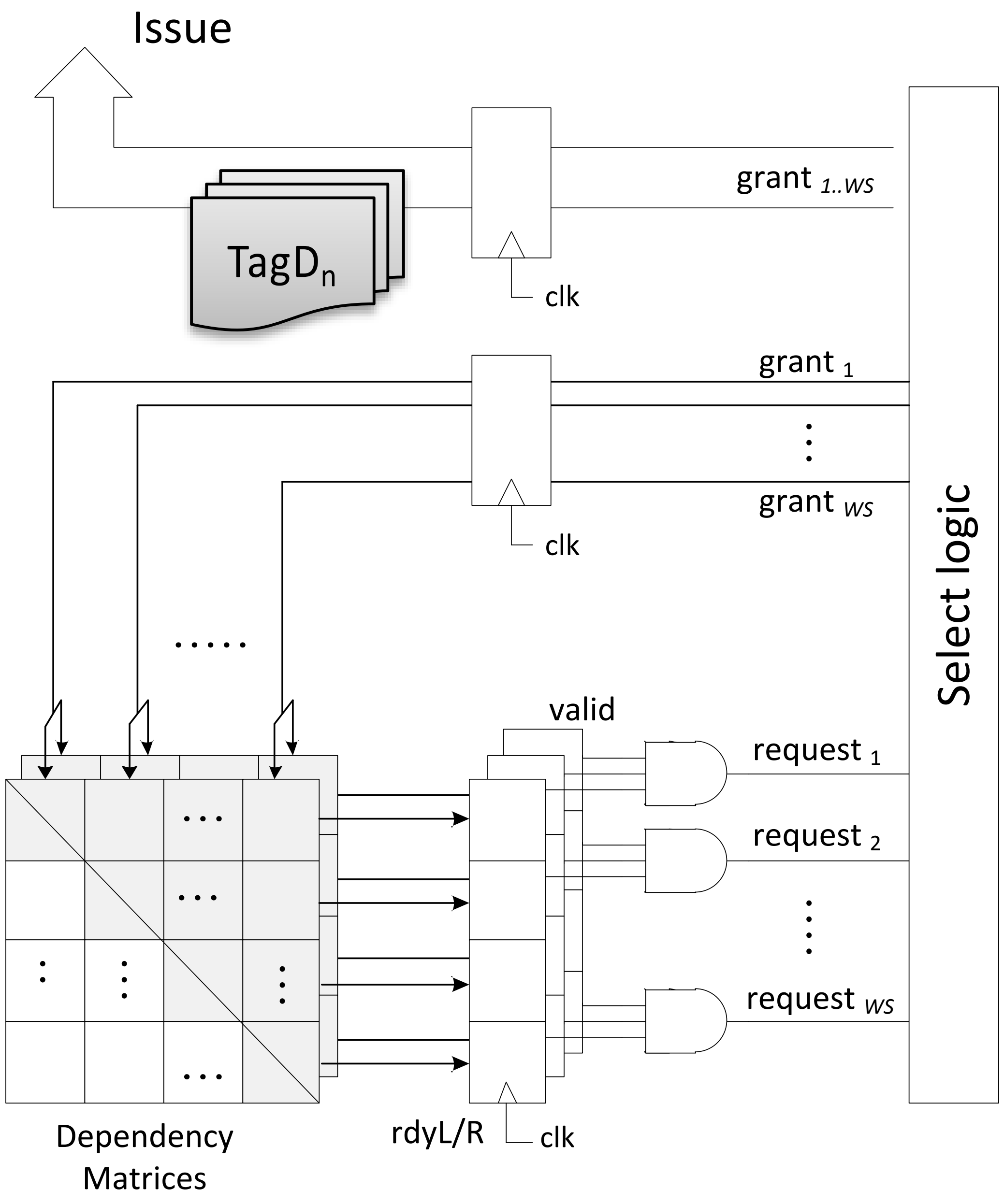


Figure 3.5 : Matrix Scheduler

Decentralization

The decentralization explained in section 2.5 can be applied to matrix scheduler also. If the instruction window is decentralized to *s* subwindows with issue width *IW'* and size *WS'*, each of the dependency matrices can easily be segmented to *s* partial matrices with *WS'* rows and *WS* columns. If the instruction window is subdivided typically in integer, load-store (LS) and floating (FP) type instruction windows, each with a size of *WS'*, dependency matrix will be divided in 3 partial matrices each with *WS'* rows and *WS* (equals to *3×WS'*) columns. And each of these partial matrices can be thought of as 3 minor matrices, one for each of integer, LS and FP each with the size of *WS'* × *WS'*. Therefore, the complete matrix configuration can be thought of as 9 minor matrices, as shown in Figure 3.6. The advantages of decentralization stated previously are also applicable in this scheme.

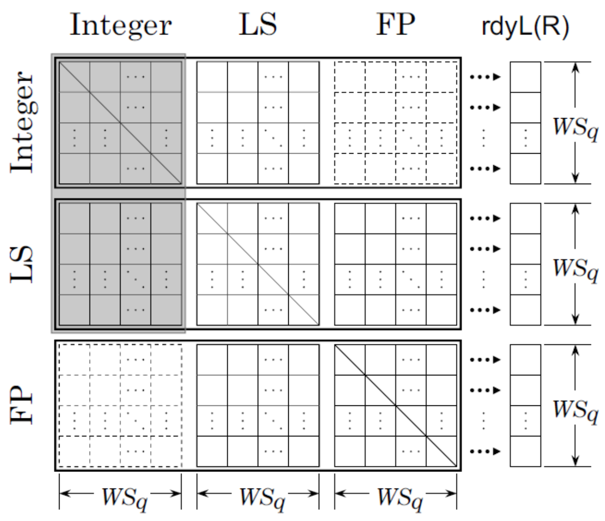


Figure 3.6[‡] **: Decentralization of matrix**

3.2 Select Logic

As explained in section 2.3, select logic is a circuit which selects a limited number of instruction for execution, out of multiple instructions whose dependencies have been resolved and they are ready for execution.

In the next sub sections we explain a few major available techniques to implement select logic.

3.2.1 Cascaded

In this method, *IW* blocks of select logic, each of which selects provides one grant, are cascaded to give *IW* instructions for issue[1,3]. A select logic implemented using

‡ Borrowed from [1]

cascaded method is as shown in Figure 3.7(b)

A single arbiter of the hierarchy is usually made up of a hierarchy of arbiter cells. Generally one such cell selects 1 out of 4 requests, and also forwards a combined request (logically ORed requests) for next cell in hierarchy. Block diagram of a single arbiter is shown in Figure 3.7(a).

Processing inside each arbiter cell is performed by 2 phases:

Forward propagation : A combined request from the lower level hierarchy cell is propagated to higher level hierarchy cell, i.e. going down the tree.

Backward propagation : The grant signal is propagated up the tree with only one of the 4 grant signals being asserted.

A multiple grant select logic uses these hierarchical arbiters and cascade them together to give multiple grant signals. As shown in Figure 3.7(b), request signal to the n^{th} arbiter are derived from the requests up to previous *n-1* blocks, by masking the requests that were granted by the previous arbiters. Thus n^{th} arbiter can select the n^{th} request and provide the n^{th} grant. Total delay of such cascaded arbiter method is given by $O(\log_4 WS \times IW)$. In a superscalar microprocessor, it is well proved that moderately increasing the *IW* gives improved IPC. However, with the cascaded scheme, performance degrades sharply with increase in *IW*. As *IW* number of arbiter blocks are needed to be cascaded in series. If *IW* becomes more than 2, the net data path delay of select logic becomes too large.

This method is not cyclic, i.e. it gives priority to a fixed location of all the available inputs. Now, if the instruction window is in wrap around configuration and we want to select the oldest instruction in the window, we need to shift the request signals till head request comes on the priority location of the select logic, and again shift back the grant signals for same amount to use it properly in instruction window. This process is adds a crucial amount of delay in cascaded method. Hence, cascaded technique may only be implemented where position priority is static.

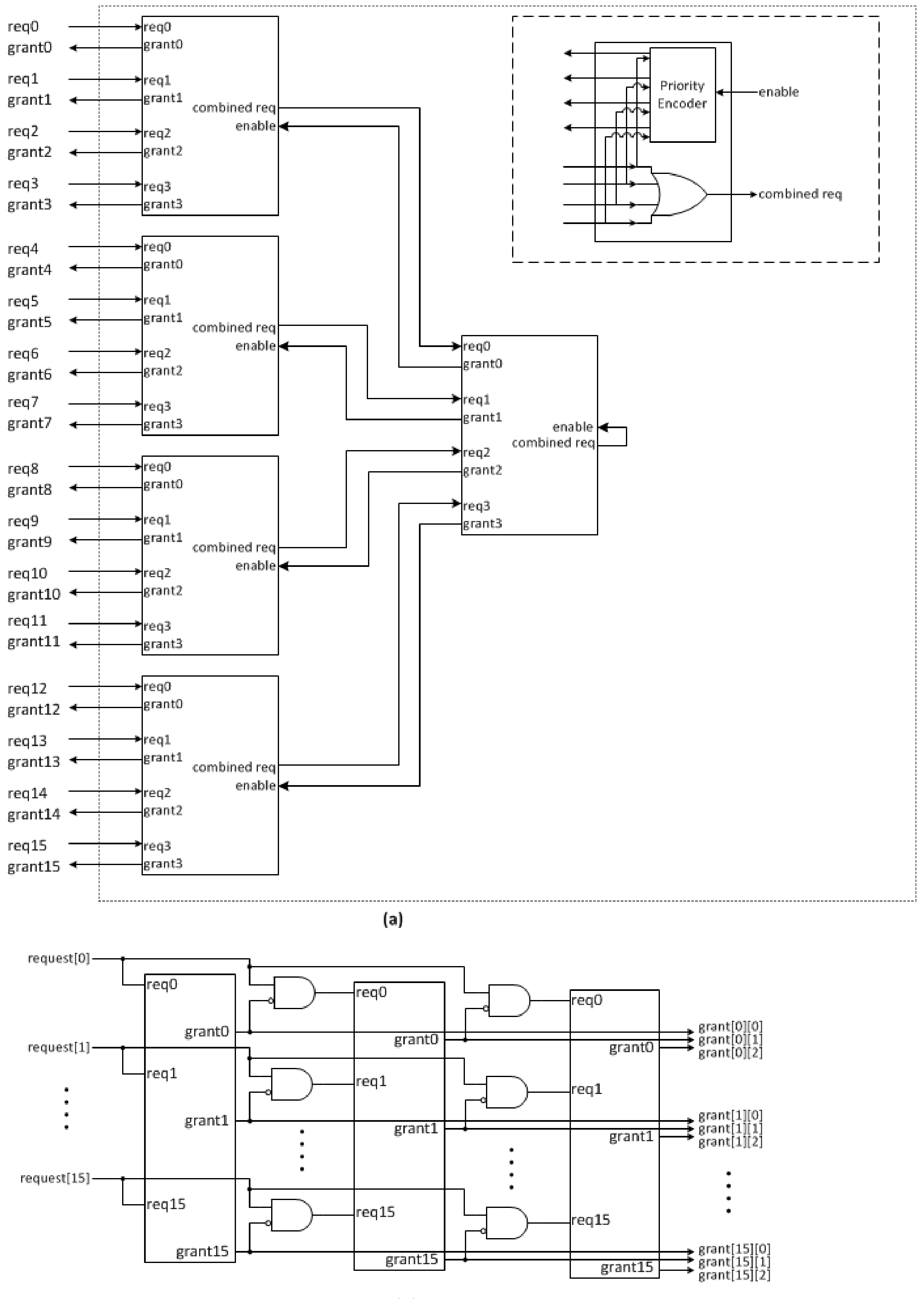


Figure 3.7 (a) Block diagram for Select Logic for single grant out of 16 requests. (b) Block diagram of Cascaded Select Logic

3.2.2 Prefix Sum thicket

This technique's critical data path delay does not vary as much with the change in IW, as with case of cascaded select logic. In this technique, all the request signals which are at higher priority than itself are added for each grant signal, i.e. a prefix sum is calculated. If this sum is less than the issue width, a grant is provided if the corresponding request is available. Total delay is given by $\log_2 WS$ delay of additions. One such prefix sum thicket for *WS* of 16 is shown in Figure 3.8.

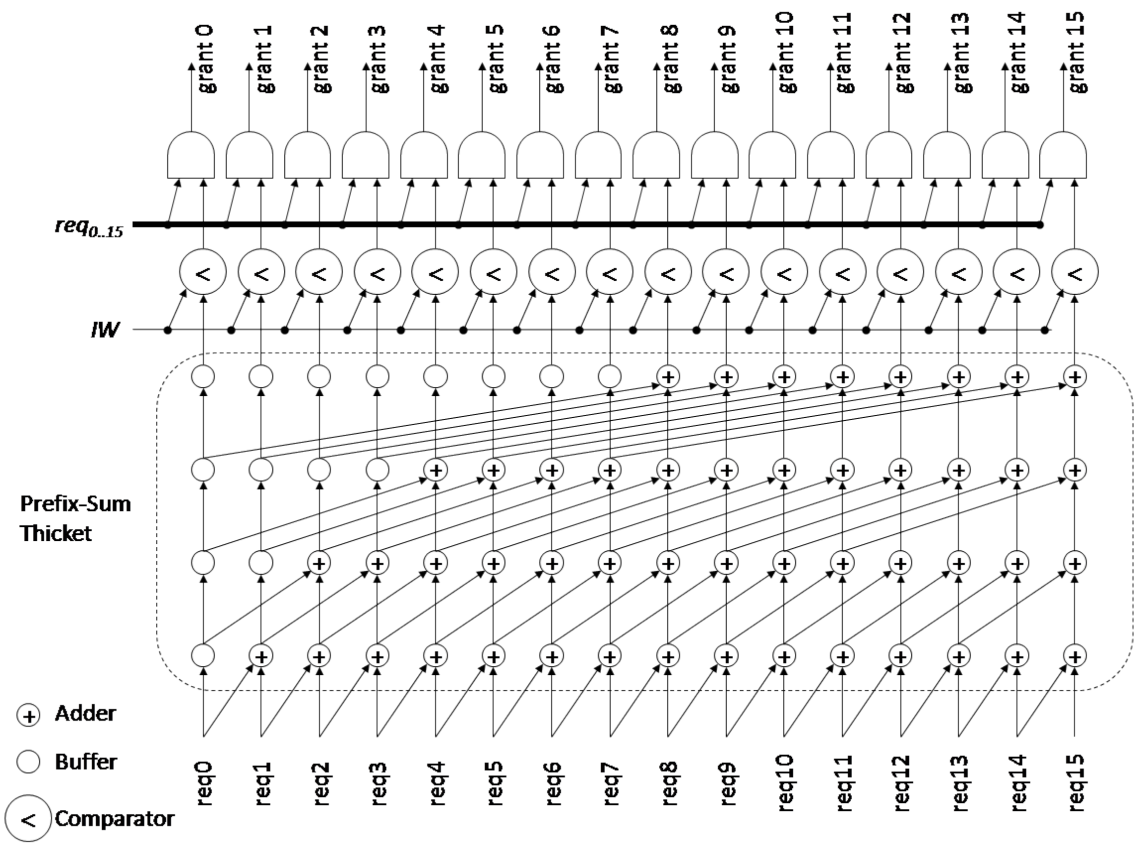


Figure 3.8 : Select Logic based on Prefix-Sum Thicket

The additions in this circuit can saturate at *IW*. Also, the circuit can be optimized by using a method explained in [7]. We propose a different optimization based on customized gray code. It is explained in section 4.2

Prefix sum thicket shown in Figure 3.8, is based on fix priority request signals, however to use floating priority location requests, a different approach towards prefix sum is implemented by [4] and is explained in section 3.2.3

3.2.3 CSP circuit based

Henry et. al. [4] gave techniques to implement select, wakeup logic and few other processor components using *Cyclic Segmented Prefix* (CSP) circuits. This technique is especially helpful with processors having wide instruction window and wrap around reorder buffer. It enables cyclic reuse of the reordering buffer with new instructions continually entering the buffer and taking up the place of the oldest, retiring instructions, without the overhead to compress instructions to the beginning of the reordering buffer.

It exploits the sequential ordering of instructions in a wrap-around reordering buffer and can attach cyclic segmented prefix (CSP) circuits to the reordering buffer. In next sub sections CSP circuit based select logics with different datapath delays are explained.

3.2.3.1 Linear gate delay

This is the simplest of all the CSP circuit configurations, CSP blocks are connected one after the other in cyclic fashion. Figure 3.9 shows a scheduler made up of CSP circuit in linear gate delay configuration.

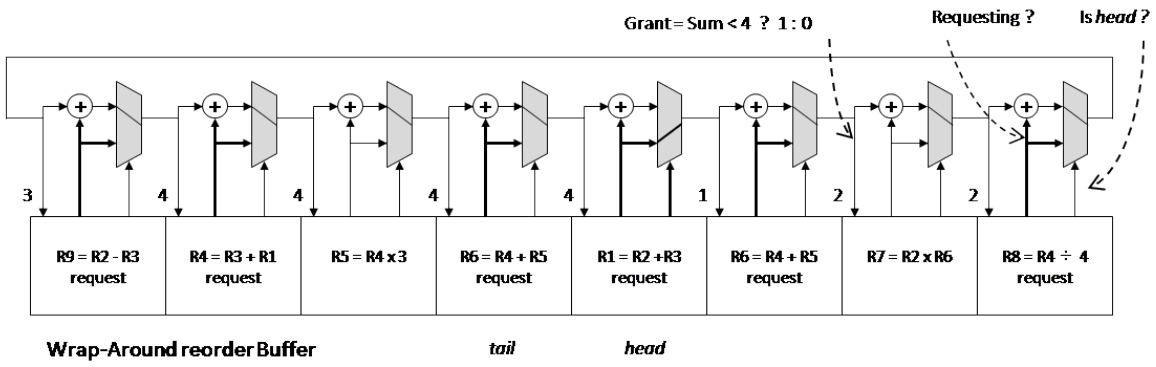


Figure 3.9 : Block diagram for Linear gate delay implementation of CSP circuit for Select Logic. Here reorder buffer entries are 8 and *IW* is 4

It generates *IW* number of grant signals starting from the oldest requesting instructions of the wrap-around reordering buffer. For each buffer entry, the select logic simply returns a sum, of all the older instructions requesting to be scheduled (sum can saturate at *IW*). A requesting entry is acknowledged with grant, if the corresponding sum is less than *IW*. The ripple carry addition starts from oldest instruction using the *head* signal, which when set, cuts the addition loop and initiate the entry with the *head* instruction's request, which is then successively added to give sum of requests up to each next instruction. Delay of such circuit is simply of the order of WS adder and WS MUX delays.

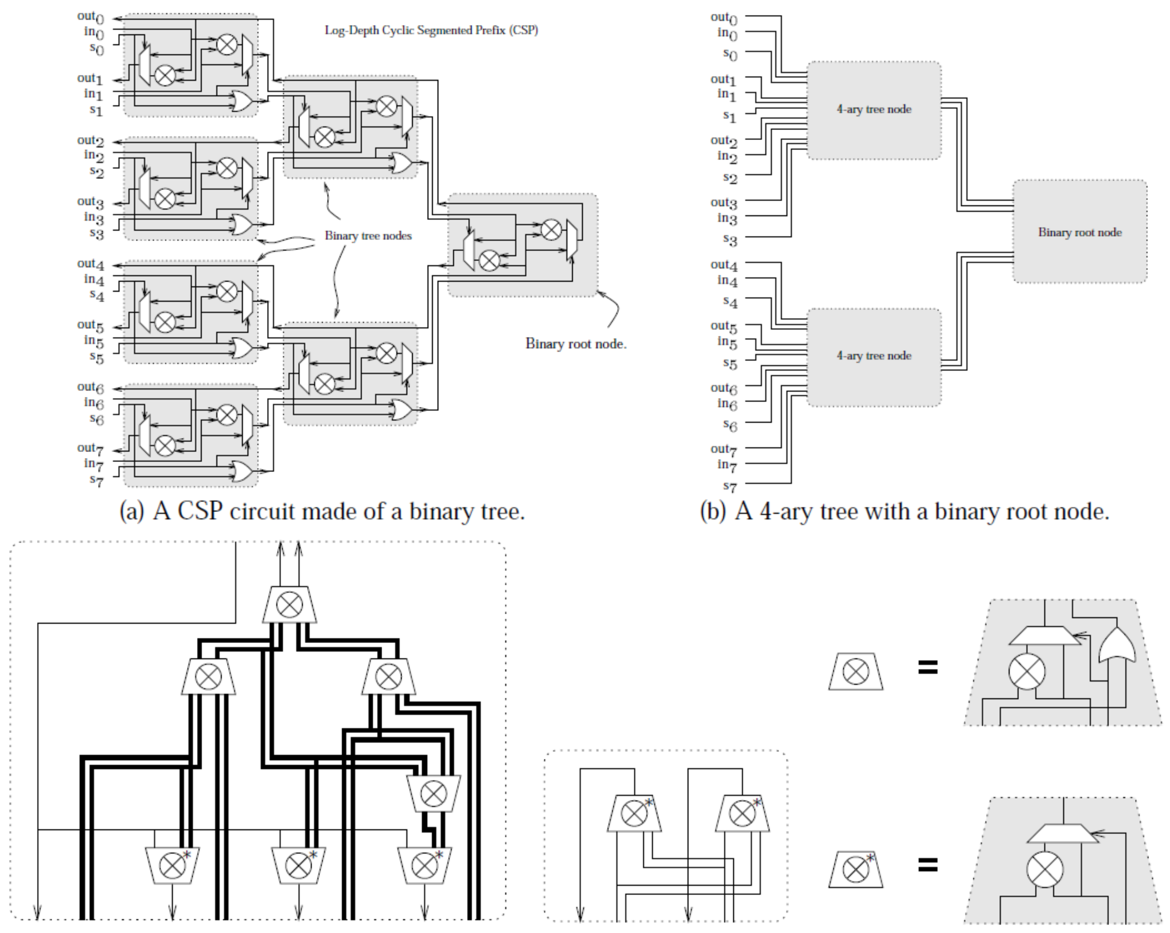
3.2.3.2 Logarithmic gate delay

Logarithmic gate delay configurations of CSP circuit can significantly reduce the critical path delay if compared to the linear gate delay implementations. A few implementations of CSP circuits having logarithmic gate delay are described in this subsection. These CSP circuits implement the same function as the linear gate delay circuit in previous section. However, the linear gate-delay CSP circuit explained earlier applies the addition operator in-order to successive input requests, on the other hand logarithmic gate-delay CSP circuits rely on the associativity of the request additions, by applying the addition in parallel to contiguous subsets of the inputs. Theoretically, they all have $O(\log_2 WS)$ order delays due to similar hierarchy level of gates, but they have different circuit areas and wire delays. A few major implementations of such log delay CSP circuit based select logics are described next.

A) Binary tree

As the name suggests, a binary tree based CSP circuit is a binary tree of multiple CSP circuit nodes, which are connected together to form a tree structure. The root node of the tree structure is modified to return the propagating sum of requests down the path to the leaf nodes. One such tree structure with total 8 request signals is shown in Figure 3.10(a). While going down the tree, each node of the tree takes two requests and their select signals then adds the requests and propagate it forward if the second select signal is not high (i.e. if it's not the *head*), else propagate the second request signal. This node differs from the linear gate delay implementation, as it also intakes an upward propagating request signals which carries the value for the total sum of the requests, which are not calculated up to that node and up to that level of hierarchy in forward path. Now as explained in previous section depending on the total sum for each request, grant signal can be easily calculated.

For WS requests, total gate delay will be given by $(2\log_2 WS - 1)$ addition circuit delay and the same amount of MUX delays. These delays are due to $(\log_2 WS - 1)$ delay in the going up the tree and $\log_2 WS$ delay in the going down the tree, for the addition circuit delay as well as the MUX delay.



(a) A CSP circuit made of a binary tree. (b) A 4-ary tree with a binary root node.

(c) 4-ary tree node. (d) Binary root node. (e) "Switched operators".

Figure 3.10§ : (a) CSP circuit implemented as a binary-tree. (b) A 4-ary tree with a binary root node. (c) A node of 4-ary tree. (d) Binary root node. (e) The switched operators used in (c) and (d).

B) 4-Ary tree

An improvement on the binary tree circuit is a 4-Ary tree. Structure of such a tree for 8 requests is as shown in Figure 3.10(b). Here 4 requests are taken into account together for a single node's input. This node provides sum of these 4 inputs to the next level of node in hierarchy. Gate delay for forward flow remains the same as that of binary tree, but the delays for backward path are halved. Hence, the gate delay and the MUX delays are both $(3/2 \log_2 WS - 1)$.

The 4-ary tree can be generalized for other widths also, with successive delay reduction in backward paths.

§ Borrowed from [4]

C) Cyclic prefix sum thicket

In this technique a thicket of trees is used. It is similar in working as the prefix sum thicket discussed in section 3.2.2, except that it cyclic, and consist of CSP circuit. It uses the *head* location to work as cyclic prefix sum thicket.

In every level of the hierarchy *WS* additions are made, compared to successive reduction of number of addition by a factor of 2, in each level of hierarchy in the Binary tree. This causes increase in circuit area, but as shown in the Figure 3.11, only $\log_2 WS$ additions and MUXs are required for each grant. It greatly reduces the gate delay, but this reduction in gate delay comes at the cost of increased wire delay for some grants signals. Yet it performs better than any other implementation of select logic briefed in this report. It can be further optimized by using our proposed custom gray codes for addition. This optimization is discussed in sub section 4.3.1.

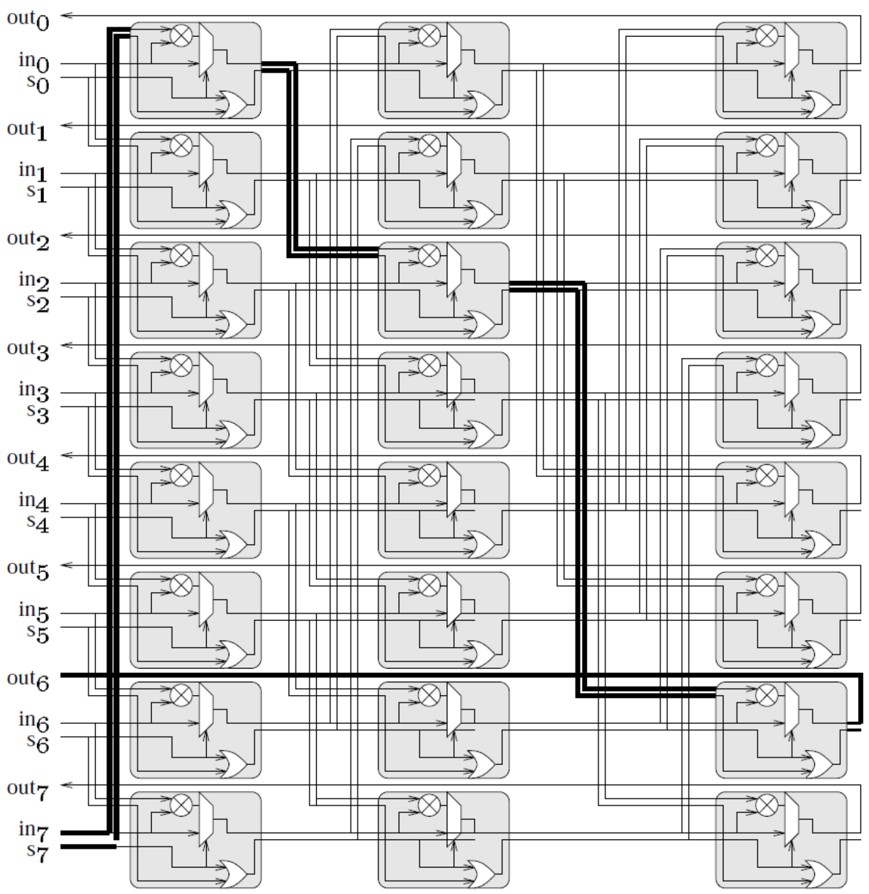


Figure 3.11[**] **: Cyclic Prefix-Sum Thicket or CSP circuit based Prefix-Sum Thicket. Longest path is highlighted.**

[**] Borrowed from [4]

Chapter 4 Implementation on FPGA

We implemented and tested different wakeup and select logic techniques, which were explained in Chapter 3, and found that Matrix Scheduler based wakeup logic and cyclic prefix sum thicket based select logic performed best among others. (Refer to chapter 5 for their performance evaluation)

To achieve high performance for instruction scheduler implementation on FPGA, we optimized and modified the original techniques for instruction scheduler, taking into account the architecture specialties and restrictions of FPGA. The proposed implementation tries to reduce the critical data path delay of Instruction Scheduler by making intelligent use of FPGA resources. The proposed modifications are explained in section 4.2 and 4.3

In order to properly understand the performed modifications and optimizations, prior knowledge of FPGA architecture is required. Thus, section 4.1 gives a brief idea about a general FPGA architecture, and then the specific architecture of Xilinx® Viterx-6® FPGA, which we used in the proposed implementation.

4.1 FPGA Architecture

FPGA (Field Programmable Gate Array), is a programmable logic device which consists of an array of programmable logic blocks of potentially different types. It includes general logic, memory and multiplier blocks which is surrounded by a programmable routing fabric that allows blocks to be interconnected to implement a digital logic. This array is surrounded by programmable input/output blocks, that connect the FPGA to outside world. One such representation of FPGA architecture is shown in Figure 4.1.

The specific design details within each of the main functions (logic blocks, programmable interconnect and programmable I/O) varies among vendors. Also, different vendors use different technology to implement the FPGA. main technologies are SRAM, Flash/EEPROM, Anti-Fuse. In this report, we focus on SRAM cell technology based FPGA from Xilinx.

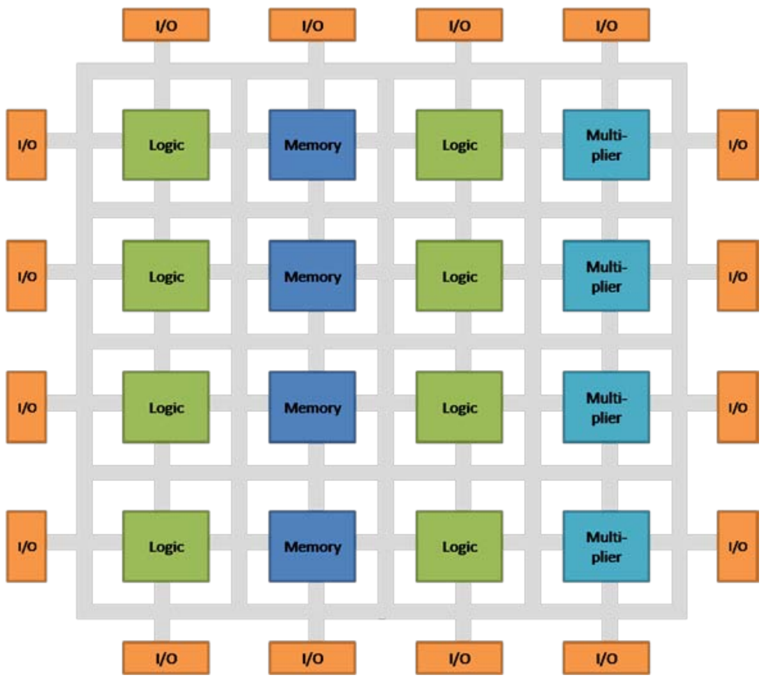


Figure 4.1 : Basic FPGA Architecture

The key component of FPGA is a programmable logic block. which comprised of a N (typically 3 to 6) input Look Up Table (LUT), a register that could act as flip-flop or a latch, and a multiplexer, along with few other elements, like arithmetic and carry chains, distributed RAM, and shift registers etc.

Figure 4.2, shows a simple programmable logic block to understand the working of FPGA.

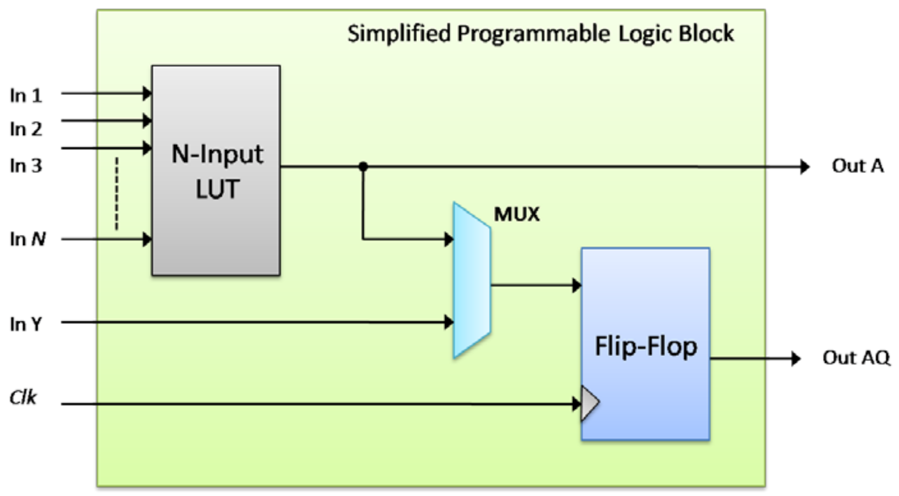


Figure 4.2 : Simplified Programmable Logic Block

FPGA contains large number of such programmable logic blocks which can be connected together using mesh of programmable interconnect wires, spreading all over the FPGA. By programming the SRAM cells, each logic block can be configured to perform different operations. Each register in the logic block, can work as either a Latch or a Flip-flop. Its input is provided through a multiplexer which can select either a input to logic block or the output of LUT. LUT itself can perform any digital logic function of composed of its input. Each digital function can be expressed by its truth table. This truth table is used to configure the SRAM cells of LUT to perform the given function, simply by reading the programmed value of corresponding to the combination of input. A bigger function with more inputs (more than the number of inputs of LUT) can easily be implemented by dividing the truth table according to the number of LUT inputs and combining the result of multiple LUTs together using the programmable interconnect.

For the implementation of proposed microprocessor[2], Virtex-6 FPGA from Xilinx is chosen. In next subsection, we explain some features of Virtex-6 FPGA.

4.1.1 Xilinx® Virtex-6® FPGA

††In Xilinx Viterx-6 FPGAs, Configurable Logic Blocks (CLBs) are the main logic resources for implementing sequential as well as combinatorial circuits. Each CLB element is connected to a switch matrix for access to the general routing matrix, as shown in Figure 4.3 A CLB element contains a pair of slices. These two slices do not have direct connections to each other, and each slice is organized as a column. Each slice in a column has an independent carry chain.

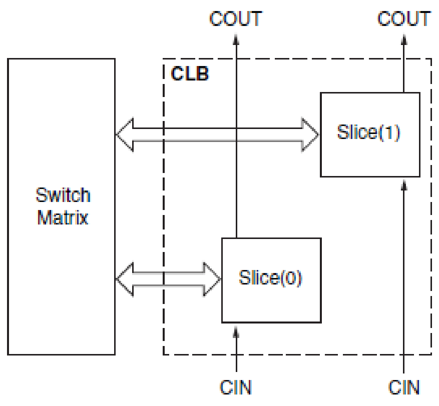


Figure 4.3 : CLB, Slices and Switch Matrix

†† Borrowed from Xilinx® Virtex-6® FPGA Configurable Logic Block User Guide (UG364)

Slice: Every slice contains four logic-function generators (or look-up tables), eight storage elements, wide-function multiplexers, and carry logic. These elements are used by all slices to provide logic, arithmetic, and ROM functions. In addition to this, some slices support two additional functions: storing data using distributed RAM and shifting data with 32-bit registers. Figure 4.4 shows one such slice (SLICEM).

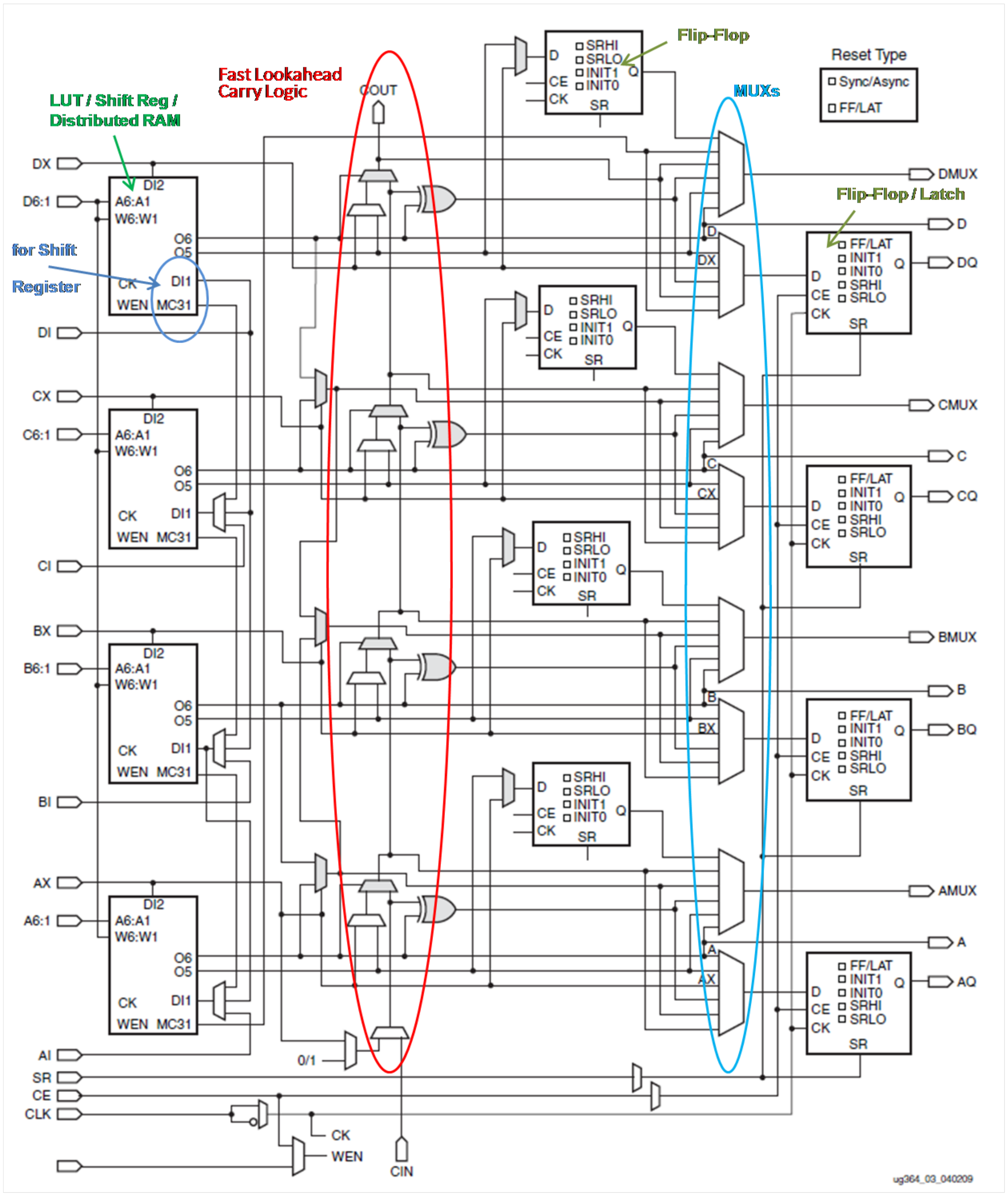


Figure 4.4 : SLICEM Architecture‡

‡ Borrowed from Xilinx® Virtex-6® FPGA Configurable Logic Block User Guide (UG364)

Look-Up Table (LUT): The function generators are implemented as six-input look-up tables (LUTs). There are six independent inputs and two independent outputs for each of the four function generators in a slice. The function generators can implement any arbitrarily defined six-input Boolean function or two arbitrarily defined five-input Boolean functions, as long as these two functions share common inputs.

Storage Elements: There are four storage elements in a slice that can be configured as either edge-triggered D-type flip-flops or level-sensitive latches. The D input can be driven directly by a LUT output or by the slice inputs bypassing the function generators. There are four additional storage elements that can only be configured as edge-triggered D-type flip-flops.

Distributed RAM and Memory: The function generators (LUTs) in SLICEM type slices can be implemented as a synchronous RAM resource called a distributed RAM element. RAM elements can be configured to implement RAMs ranging from single-port to quad-port of different sizes.

Shift Registers: A SLICEM type function generator can also be configured as a 32-bit shift register without using the flip-flops available in a slice.

Multiplexers: Function generators and associated multiplexers can implement the up to 16:1 multiplexer using up to 4 LUTs

Fast Lookahead Carry Logic: Dedicated carry logic is provided to perform fast arithmetic addition and subtraction in a slice. A CLB has two separate carry chains. The carry chains can be cascaded to form wider add/subtract logic.

4.2 Matrix Scheduler

The Matrix scheduler technique [1], as explained in sub section 3.1.4, uses RAM for the dependency matrices. However, due to the reasons given below, a RAM based matrix scheduler is not feasible on current FPGAs.

1. **Multiple dispatch :** There are multiple simultaneously dispatched instructions in the under consideration microprocessor, thus in the dependency matrices, multiple

wordlines need to be updated simultaneously with the dependency bits for instruction. For the purpose, a multiple write port RAM is required, which is not available in current FPGAs.

2. Multiple issue : As our processor is superscalar and issues multiple instructions. Matrix scheduler must read *IW* wordlines of RAM simultaneously, which is not possible in a conventional RAM. Also, originally the grant signals from Select Logic were directly used as wordlines of RAM. However, in conventional RAM, address decoders are inbuilt to assert corresponding wordlines. Thus, grant signals cannot be used directly as wordlines. They first need to be encoded as individual addresses, which can be used as address inputs of multiport RAM, which again gets decoded as active wordlines. This will cause an un necessary increase in data path delay, and above all with $IW > 2$ multiple read port RAM is required which is not available in FPGA.

Neither the block RAM nor the distributed RAM can perform above the multiport operation required as above. Finally, the only option left is to use a matrix of flip-flops to work as multi-read, multi-write dependency matrices.

In the original proposal of matrix scheduler, grant signals from select logic were used in the modified RAM's wordlines to directly provide the bitwise OR of the columns of issued instruction. However, now when the matrices are implemented using flip-flops, in order to generate the rdyL/R bits, we additionally need to perform this operation.

To select the appropriate columns for bitwise OR operation, circuit requires a *IW* number of *WS-to-1* multiplexers for each row. An easier approach to such operation is to bitwise AND each row of matrix with grant signals and take logical OR for each row of resulting matrix.

Such bitwise AND and logical OR operation is required on each row of each dependency matrix to generate their ready bits column vectors.

4.2.1 Proposed changes

As the advantages of using the (modified) RAM is already lost and we need to read a complete row to generate a ready bit, we can change the dependency matrix composition to achieve optimized results for flip-flop based memory. In our proposed technique, we use an integrated matrix instead of different matrices for each operand. It stores dependency bits of all the operands together. Dependency relation among cells is still the same as described in the original technique[1], except that each cell now represents the dependency of either operand. Such a dependency matrix is shown in Figure 4.5

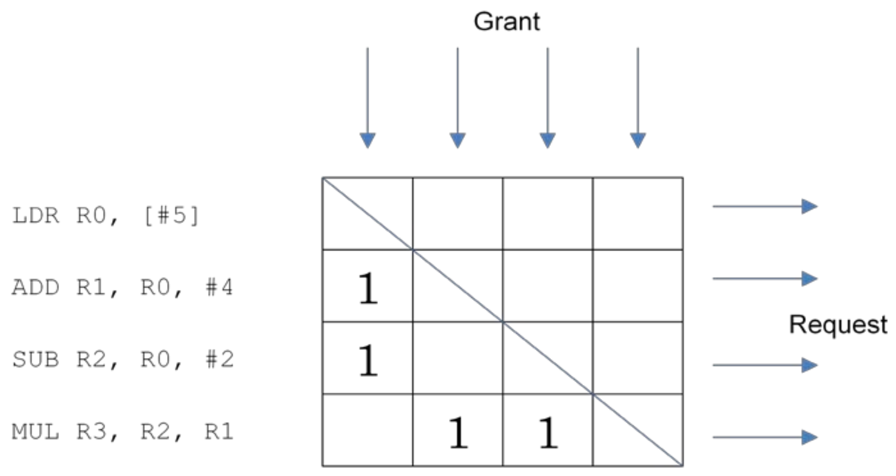


Figure 4.5 : Single Integrated Dependency Matrix

Now as all the operands' dependencies are jumbled up in a single matrix, the rdyL/R bits for each operand must be replaced by a single logic which determines the readiness of all the operands combined i.e. in each cycle, checking whether all the producer instructions for any consumer instruction have been issued or not.

As, it is not necessary that all the producer instructions for any consumer instruction be issued at once, we need to keep track of the issued instructions. This is done by ORing the successive grant signals and saving in registers.

Matrix update: The proposed processor implements renamed trace cache[8], which uses the distance between consumer and the producer instructions as a parameter for dependency. We use this distance based dependency information to update the matrix. Newly dispatched instructions are required to update in matrix consecutively from the tail location. Distance of dependent operand from its producer is read and appropriate dependency bit can be easily generated by subtracting the producer's distance from tail, and shifting 1 by that amount. Dependency bits for other the operands are similarly obtained, and written to the matrix simultaneously.

Wakeup: While reading the matrix, for each row only the required producer bits are masked of all the available producer bits. If this masked result has all the 1s as required by the consumer, a request is generated. This process is accomplished by bitwise ANDing a row with accumulated grant registers for masking and the result is compared with the row itself. If it is the same, corresponding request is asserted. This operation can be understood with the help of Figure 4.6.

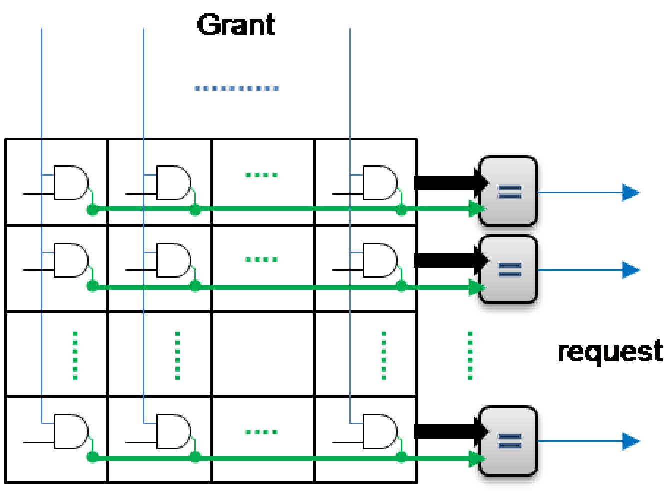


Figure 4.6 Generation of request signal

As a single LUT can perform even complex operations on given inputs, rather than just ANDing in first step. The above operation get converted to the operation as below, and shown in figure 4.7

$$\text{Req}[n] = \text{Row}[n]\ \&\ ((\sim\text{Row}[n])\ |\ \text{grant})\ ;$$

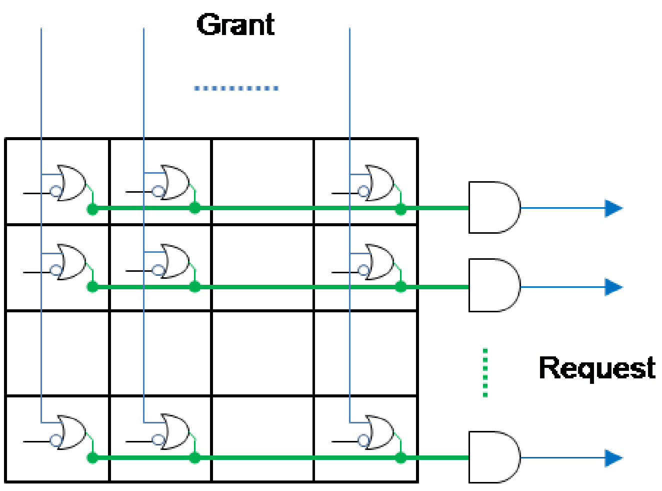


Figure 4.7 : Generation of request signal with changed equation

The last step of ANDing the Row[n] with the other half of the above equation can be optimized by the using "*Fast Lookahead Carry Logic*" available inside FPGA.

As Virtex-6® FPGA has 6 input LUTs, 3 bits from grant register and 3 bits from matrix row can be solved for the ANDing and matching operation using a single LUT. Other bits of the matrix row can be similarly be computed in parallel.

Now on the second step, outputs of each of these LUTs must be ANDed together to give the final request signal. Further LUTs can be used for the purpose, but it will result in a hierarchy of LUTs. In our case, the largest instruction window for the Load Store type instructions is a 42 instruction window. Now, a 42 bit wide matrix row needs 3 level of LUTs spread over multiple slices giving a delay of roughly 0.061×3 *ns* for LUT combinational delay and additional inter slice net(wire) delays, approximately in range of 0.3 to 0.9 *ns*. As each level of the LUT is in different slices/CLB, wire delays cost a lot more than the actual combinational circuit delay inside LUT.

Hence, we go for a better approach and used the *Fast Lookahead Carry Logic* built inside the slices. As its name suggests these carry logics are basically meant for lookahead carry generation / propagation for large adder/subtractor circuits. However, we used these elements to optimize our logic as explained next.

As each slice contains 4 LUTs, 4 results of 3 bit pairs' ANDing and comparison can be made available inside a slice. These results are used as selectors for the multiplexers of carry network. The very first multiplexer inputs are fixed to 1 and 0, and its output feeds next multiplexer's input on selection 1. Thus at any time, if any of the 4 results is 0, a 0 is selected though the corresponding multiplexer and it is propagated till the end.

The structure of our proposed wakeup logic block is shown in the figure 4.8. MUXs in the figure represent *Fast Lookahead Carry Logic.*

The Carry-in to carry-out delay (0.068ns) of a slice is almost the same as the combinational delay of an LUT (0.061ns). On the other hand, as the 4 level carry network is built inside the slice, wire delay are reduced to zero (except for the delay during carry propagation to next slice.).

The two structures for the masking and comparison process, were easily identifiable in the critical path of the circuit, shown by the mapping tool (Synopsys Synplify). These two critical paths are shown in figure 4.9 and 4.10.

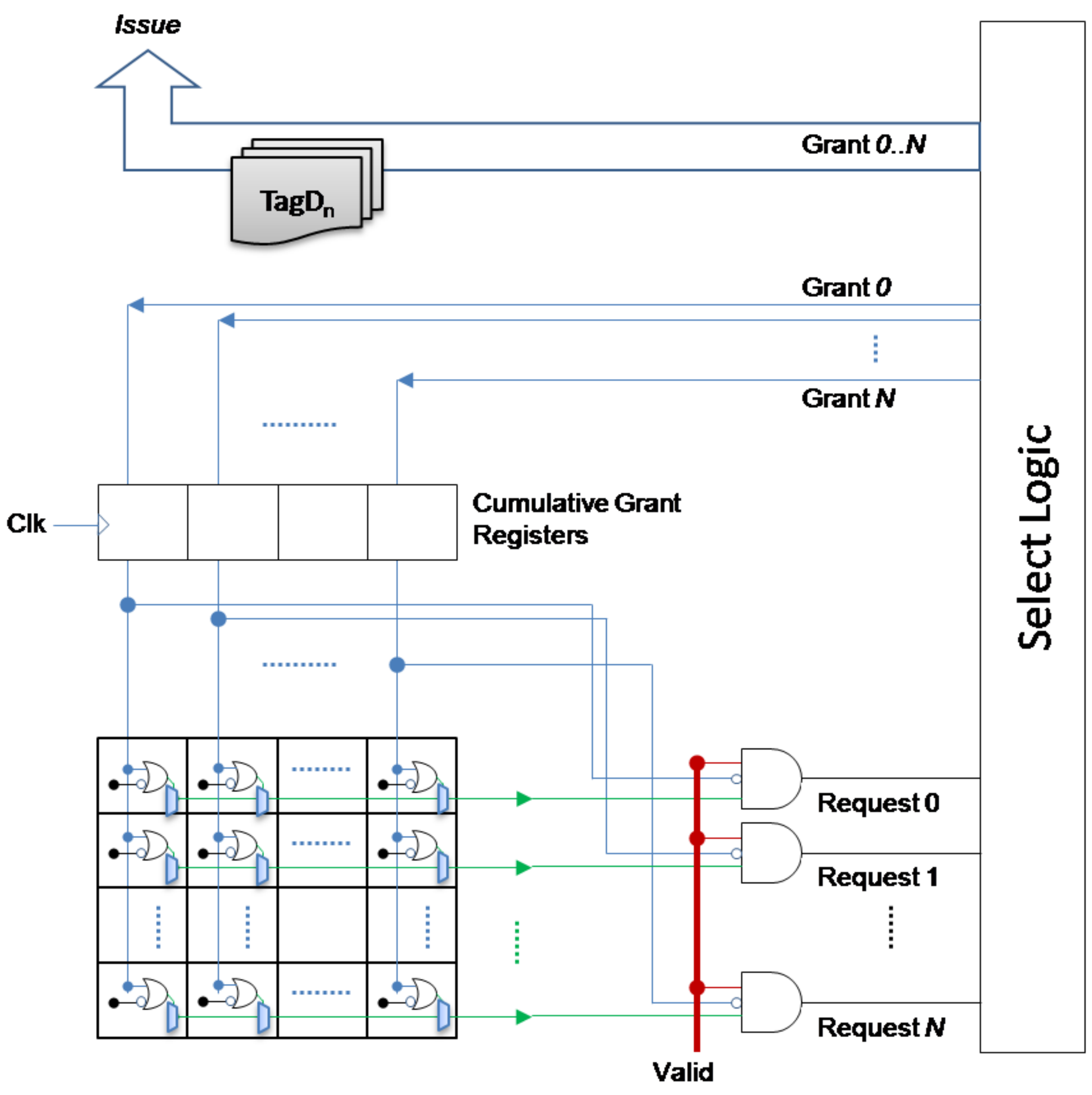


Figure 4.8 : Proposed Matrix Scheduler

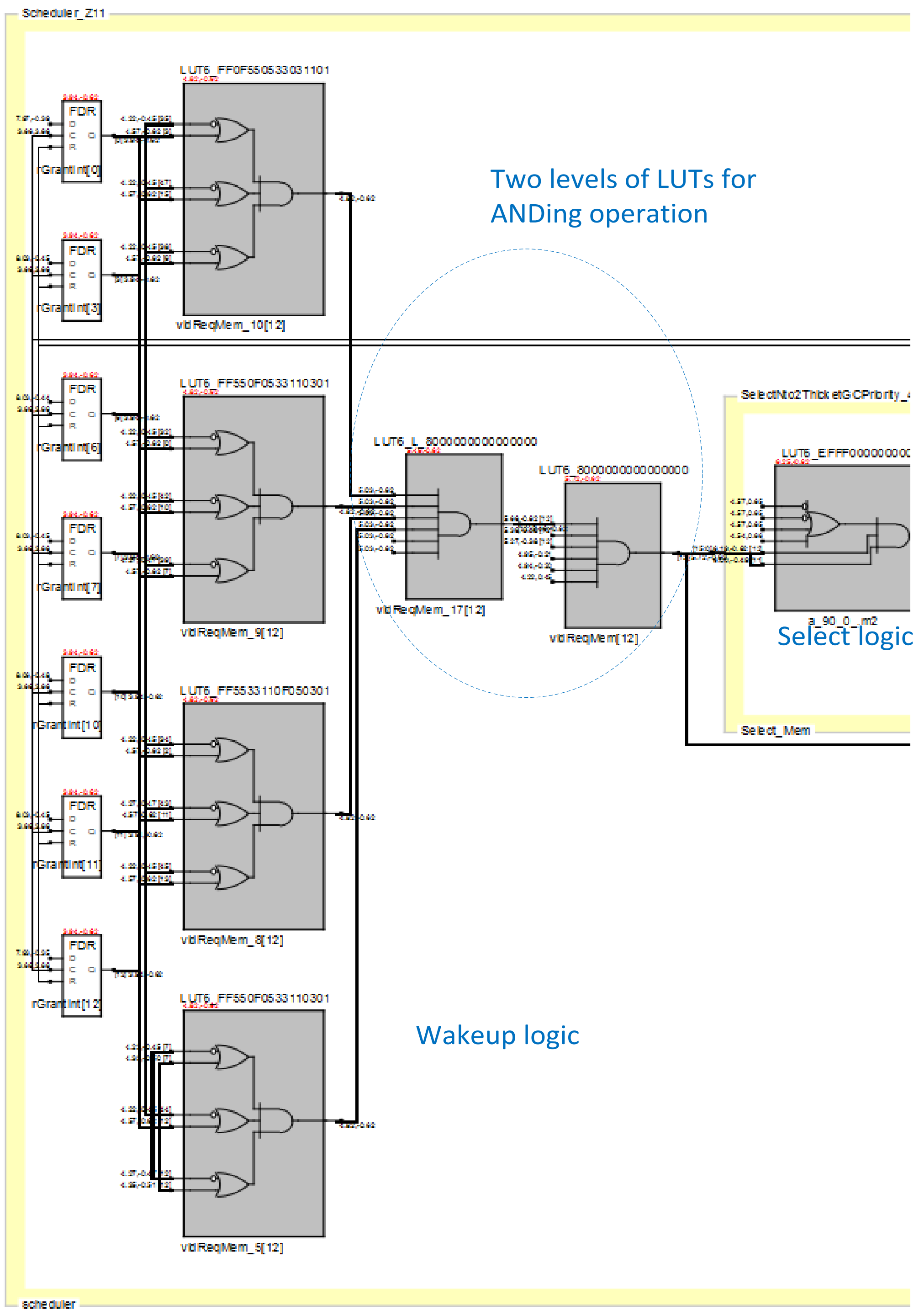


Figure 4.9 : Critical path of 48bit AND operation resulting in hierarchical LUTs

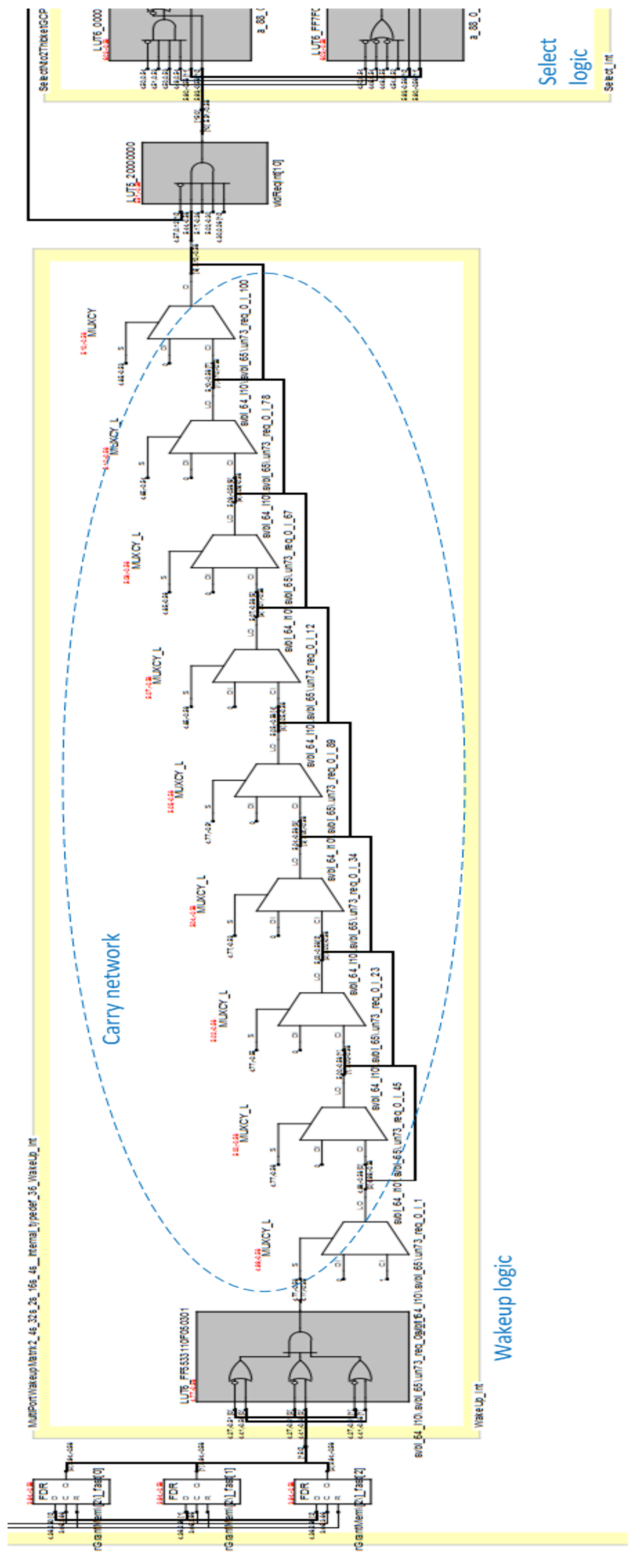


Figure 4.10 : Critical path of 48bit AND operation using Fast Lookahead Carry Logic of Xilinx FPGA.

4.3 Select logic (Cyclic prefix sum thicket)

We tried few different techniques for select logic and finally the thicket of CSP circuit (Cyclic prefix sum thicket) based implementation gave the best results.

As stated earlier in the 3.2, cascaded select logic is not good if the *IW'* is more than 2, as the delay rises sharply with *IW'*. Although in our case the *IW'* is 2, yet its performance is not as good. The CSP based circuits have different delays for each type of implementation technique, out of which the thicket type worked better than others.

In the thicket configuration, each CSP block takes tow 3 bit inputs (2 bits for previous sum input and 1 head bit) and give 2 bit saturated sum, and a combined head bit. These outputs are then read by two different CSP blocks. Hence the possibility of merging 2 or more blocks in a single LUT is almost eliminated. Hence, we leave the little task of making optimum use of LUTs on the mapping tool.

However, we propose a optimization technique in the adder circuit, explained next.

4.3.1 Optimization using custom gray codes

First we use saturated addition, i.e. add result must not exceed *IW'+ 1*. In our case *IW'* is 2, so the total sum can be saturated at 3. Hence only 2 bit adder is required with 2 bits for result and no carry is generated.

Above this, we optimized the adder by using custom gray codes so that the final checking of sums, whether they are less than or equal to *IW'*, is simplified to just checking the LSB of the sum. This helps to decrease the inputs to the final level LUT from which the grant it being provide, it enables to optimize the LUTs by merging together the calculation of last stage of thicket and checking the last sum along with the request signal. The gray codes for 2 bit adder are shown in Verilog code of Figure 4.9

```
case( {in1, in2} )
      4'b0000: Sum = 2'b00;   // 0 + 0
      4'b0001: Sum = 2'b01;   // 0 + 1
      4'b0100: Sum = 2'b01;   // 1 + 0
      4'b0101: Sum = 2'b11;   // 1 + 1
      4'b1100: Sum = 2'b11;   // 2 + 0
      4'b0011: Sum = 2'b11;   // 0 + 2
      default: Sum = 2'b10;   // 3
endcase
```

Figure 4.11 : Custom Gray codes for addition operation

From the above Figure, it is clear that if bit 0 of the sum is 1, the sum is less than or equal to 2 (IW) and a grant can be provided to corresponding request.

Chapter 5 Evaluation

This research of implementation on instruction scheduler is a part of a softcore version of the proposed superscalar microprocessor[2]. And thus focused around the parameters used in the proposed processor. Although the processor would be customizable in terms of processor components, size etc, but it is initially targeted to have the following configuration, which are of our interest in the research.

Configuration type	Value
• Base ISA	32-bit ARM
• Number of operands	3
• Issue Width for Integer type instructions (IW_{int})	2
• Issue Width for Integer type instructions (IW_{FP})	2
• Issue Width for Integer type instructions (IW_{LS})	2
• Dispatch width for each window	4
• Decentralized window size	
✧ Integer	16
✧ Floating Point	16
✧ Load Store	16

We used, System Verilog for writing the circuit, and used Synplify Mapper from Synposys Inc., and Xilinx’s Place and Route tool to evaluate our circuit.

The implementation is performed on Viterx-6 FPGA (XC6VLX760 FF1760 -2) from Xilinx. It has 6 input LUTs. Other features of Virtex-6 are already explained in section 4.1.1.

5.1 Wakeup Logic

The Critical path delay of CAM based and Matrix scheduler based wakeup logics are as shown in figure 5.1

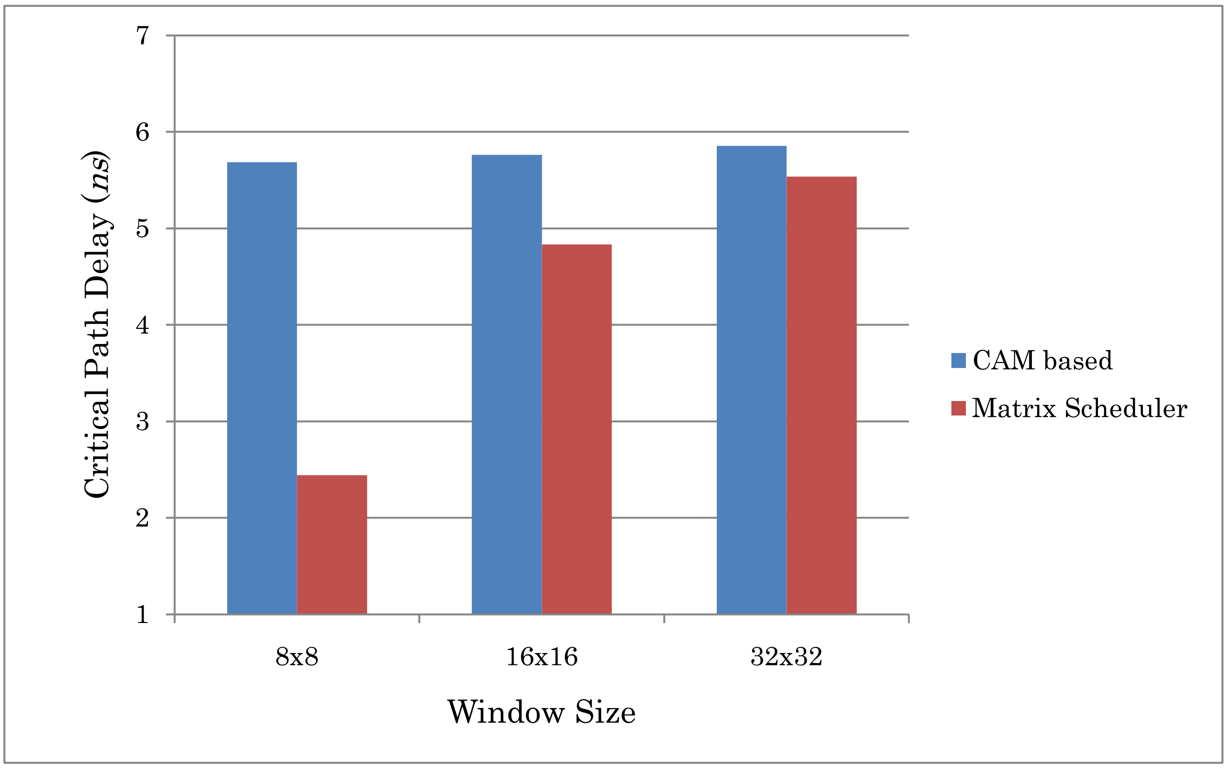


Figure 5.1 : Critical path delay of CAM based and Matrix Scheduler based Wakeup Logic

There are two interesting observations in the above figure.

A) The CAM based wakeup does not vary much with change in Window Size. This is because of the fact that all the comparison made with additional window size are happening in parallel, however after the comparison a bitwise OR is required for the all the comparison results. This comparison and bitwise OR is completed using the *Fast Lookahead Carry Logic*, whose overall delay does not vary much if the chain is in small range. Neighboring Slices can be added for even less inter slice wire delay.

B) There is not much difference between the CAM implementation and the Matrix scheduler implementation (after WS of 16x16), when they are implemented on FPGA. On the other hand [1] showed a huge difference between the critical path delay of matrix scheduler and the CAM method of Wakeup.

Figure 5.2 shows the LUT utilization of the these two circuits. As expected CAM based structure uses a lot more LUTs than Matrix scheduler.

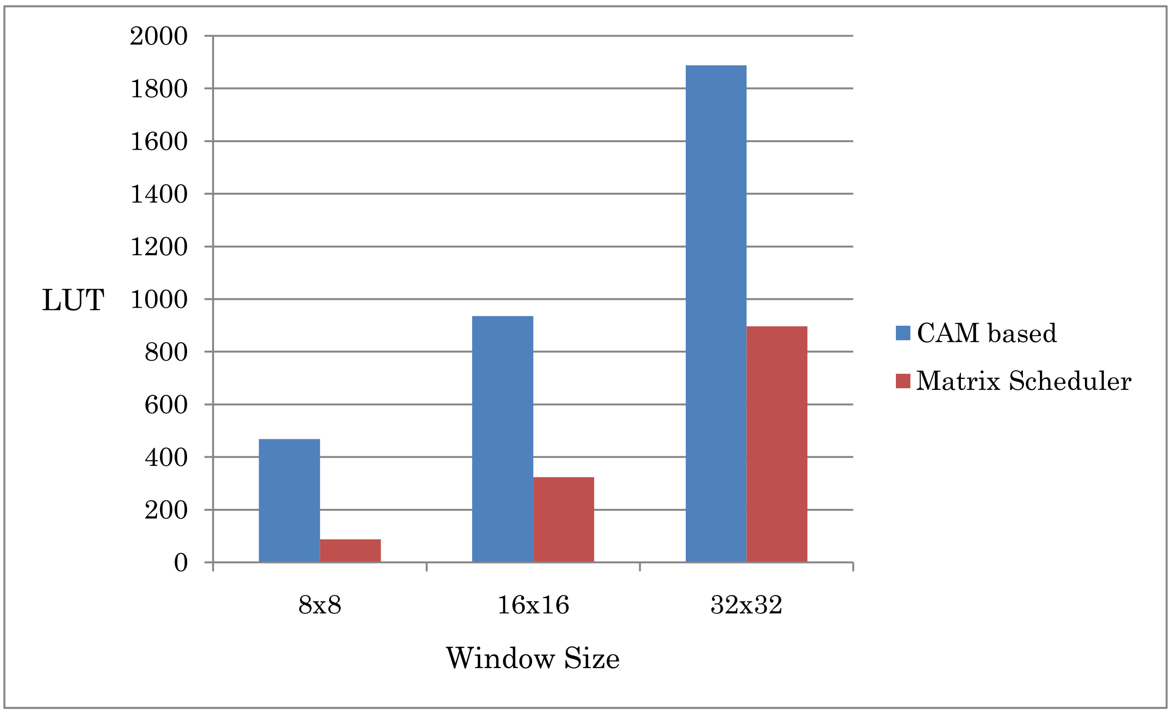


Figure 5.2 : LUT utilization of CAM based and Matrix Scheduler based Wakeup Logic

5.2 Select Logic

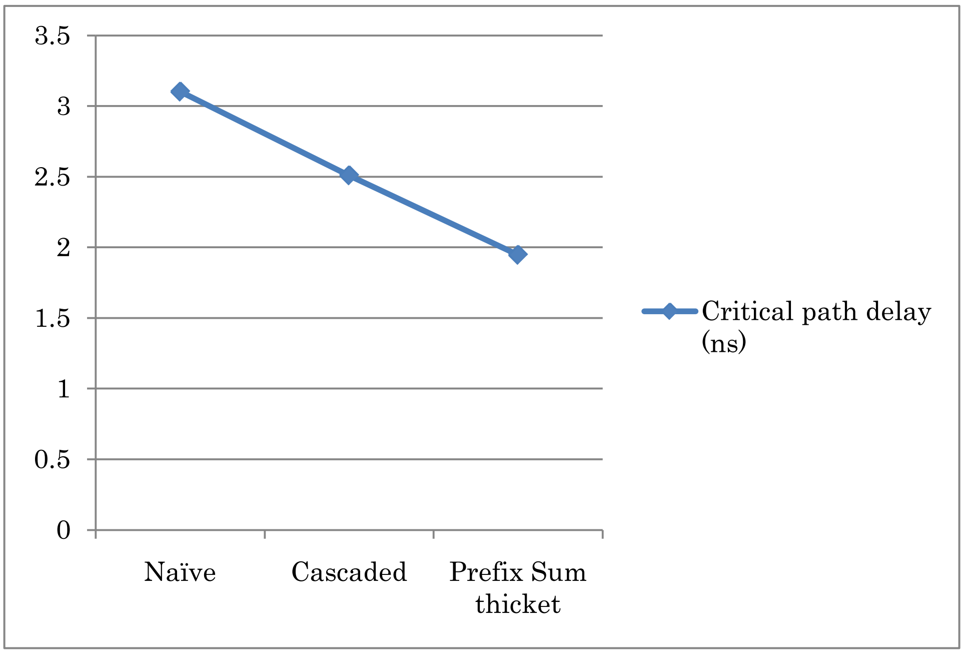


Figure 5.3 : Critical path delay of fix priority select logic techniques when *IW* is 2 and WS is 16

The Critical path delay of fix priority select logic techniques when *IW* is 2 and *WS* is 16is shown in figure 5.1 and it can be seen that the delay of prefix sum thicket is smaller than the cascaded technique, as the latter's delay increases with

$$O(\log_4 WS \times IW).$$

On the other hand, for Prefix Sum thicket its Logarithmic

$$O(\log_2 WS).$$

The critical path delay for various cyclic select logic techniques are shown in figure 5.2. It can be observed with the graph that the proposed optimization which was explained in section 4.3 gave best result.

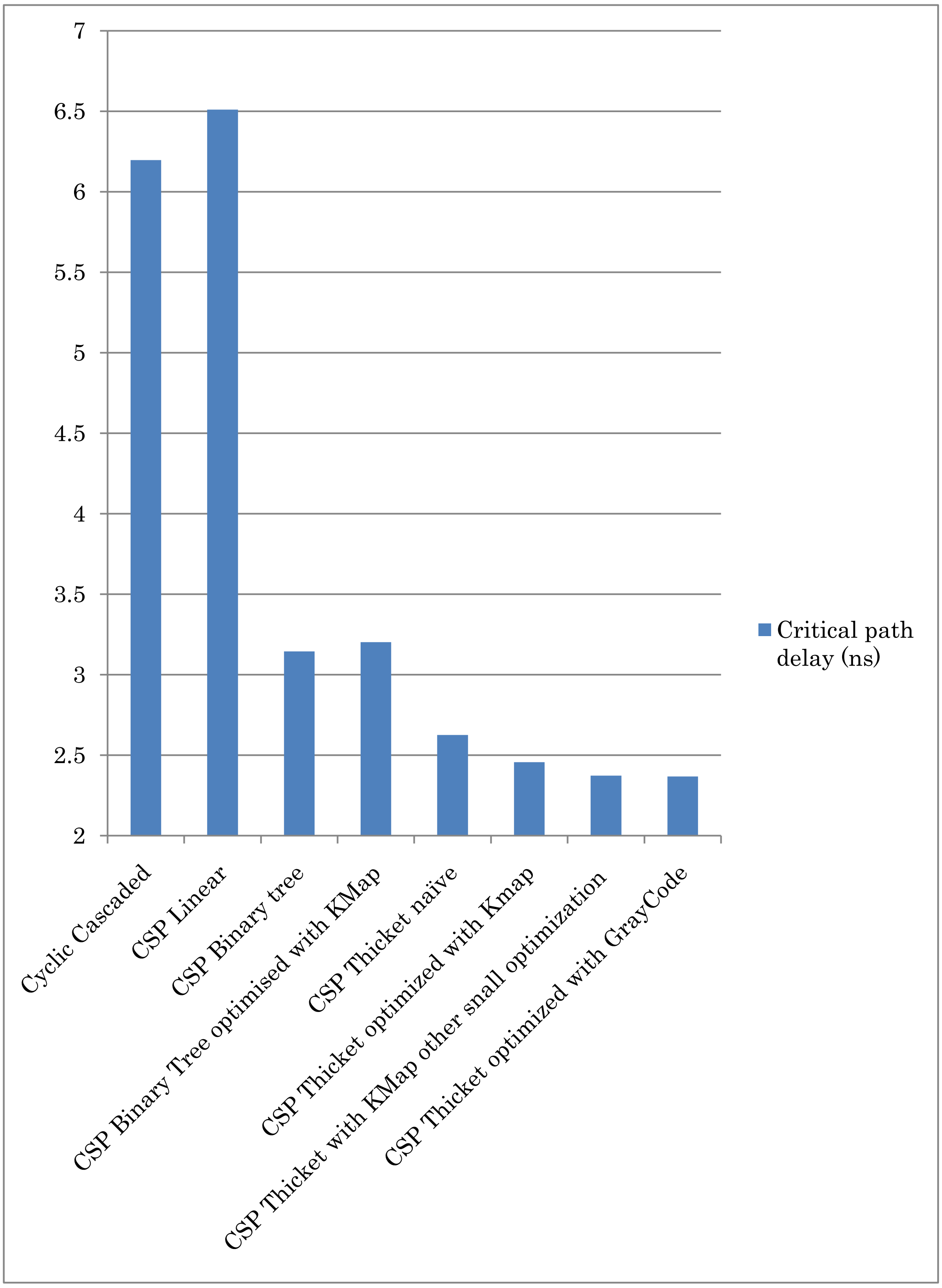


Figure 5.4 : Critical path delay of cyclic Select Logic techniques when *IW* is 2 and WS is 16

5.3 Instruction Scheduler

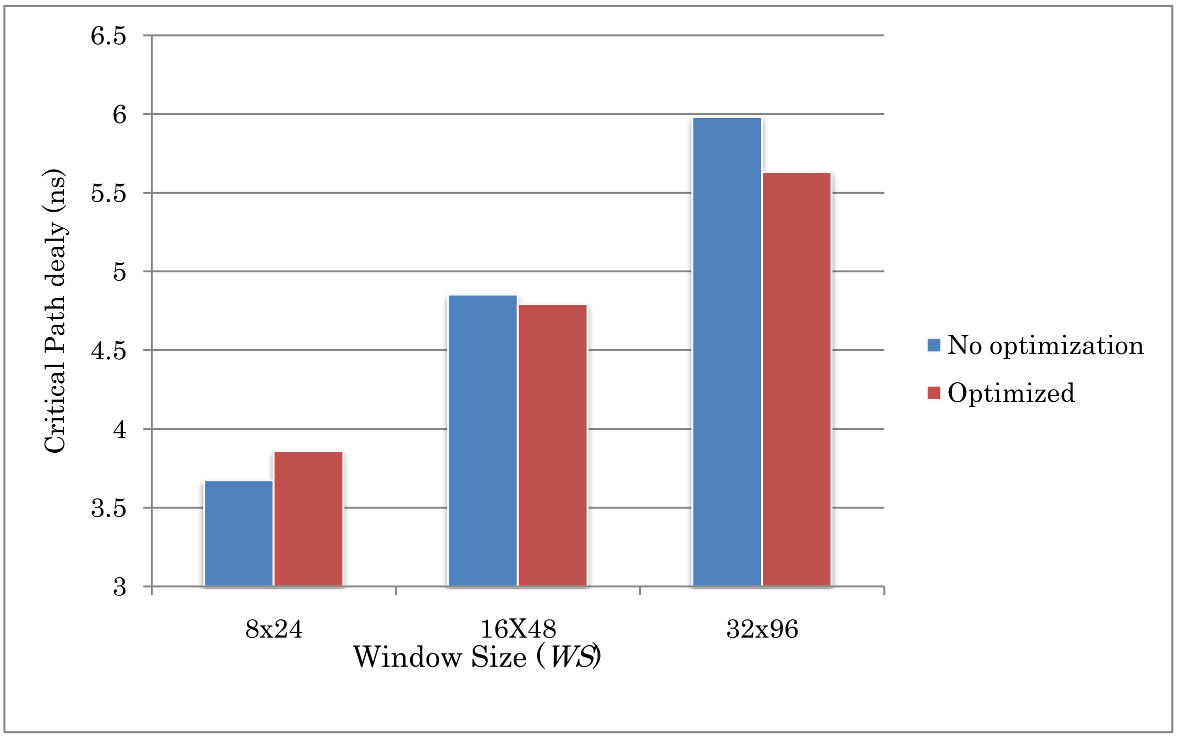


Figure 5.5 : Critical path delay of proposed implementation of Instruction Scheduler. Window Size v/s time (ns)

As shown in figure 5.3, the critical path of proposed matrix scheduler is better than the general one if window size 8 or 16. However, as window size increases further, the delay caused by large number of successive carry chain, it gives a poor performance.

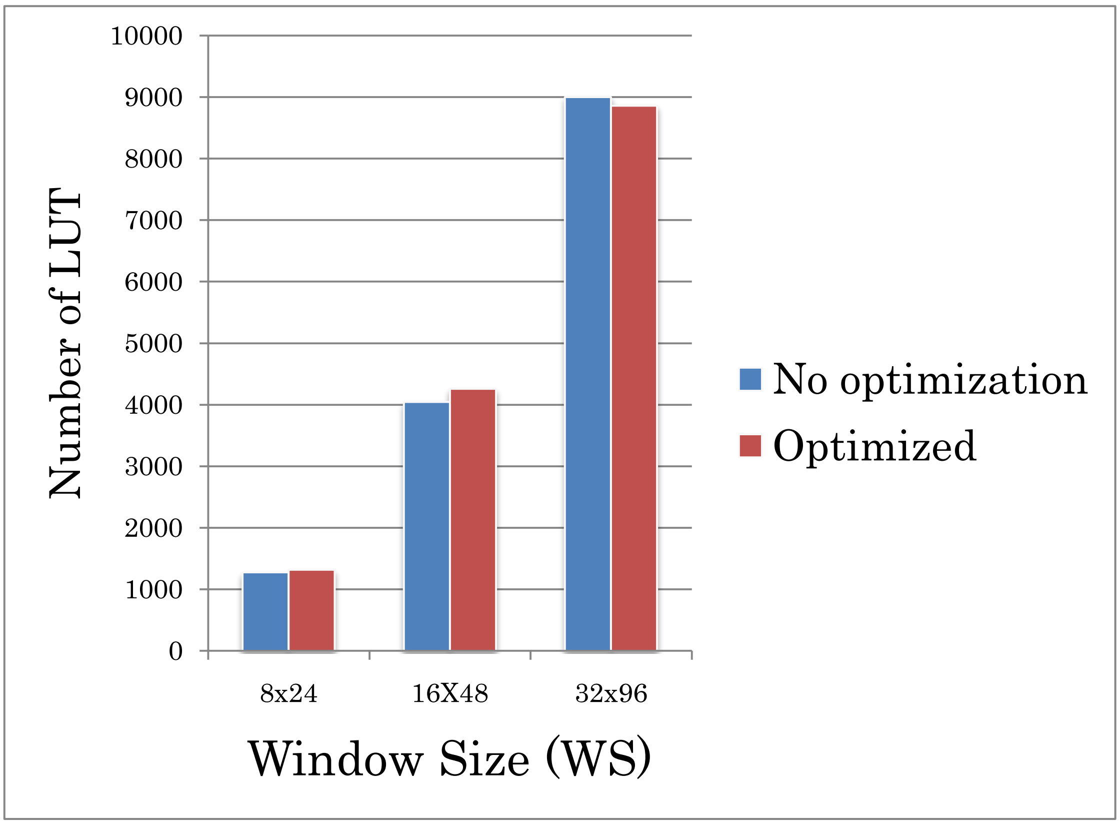


Figure 5.6 : LUT utilization of proposed Instruction Scheduler with changing WS

Chapter 6   Conclusion

In this research a scheme to efficiently implement instruction scheduler on FPGA is proposed. This implementation is a part of FPGA implementation of an area efficient out-of-order superscalar microprocessor proposed by our lab[2].

Instruction scheduler is a critical components of microprocessor which issues instructions for execution as soon as their dependencies are resolved. As its two components “wakeup logic” and “select logic” can be pipelined, it becomes the slowest pipeline stage of microprocessor. On the other hand, FPGA has a very different architecture from ASIC. It has various restrictions and special features, any circuit being implemented on FPGA must consider this fact and should be modified moderately to take advantage of FPGA architecture.

Hence, in this research we proposed the changes required in the original scheme of matrix scheduler (used for wakeup logic) in order to physically implement it on FPGA. The proposed changes include A) Using of flip-flop array of used as memory instead of the modified RAM. B) Using singular dependency matrix instead of different dependency matrix for each operand and C)a subsequently a different method to generate ready bit, compared to the original technique. Another proposal of this research is a way to decrease the critical data path delay by intelligently using an special FPGA feature of “*Fast Lookahead Carry Logic*” for request signal generation, which replaces the hierarchical LUTs for ANDing logic.

In this research different techniques for select logic were test implemented and it was found out that Prefix-Sum Thicket based implementation of CSP circuit gave the best results. We then optimized it for even better performance on FPGA by using saturated addition and also by using custom gray codes which help to merge two more LUTs together in the final stage of addition, and thus reducing critical data path delay by elimination of wire delay and also the LUT combination delay.

Finally, the proposed implementation technique make it possible to run the instruction scheduling pipe stage of processor at 208 Mhz on Xilinx Virtex-6 FPGA in comparison to 117 Mhz of the non optimized, naive implementation of same techniques for a configuration specific to the under consideration microprocessor.

References

[1] M. Goshima, K. Nishino, Y. Nakashima, S. Mori, T. Kitamura, and S. Tomita: A High-Speed Instruction Scheduling Scheme for Superscalar Processors, *MICRO-34*, pp. 225-236 (2001)

[2] K. Horio, R. Shioya, M. Goshima, and S. Sakai: Design of Area-Efficient Processor. *Symp. on Advanced Computing Systems & Infrastructures (SACSIS),* pp. 339—346 (2010)

[3] S. Palacharla, N. P. Jouppi, and J. E. Smith: Quantifying the complexity of superscalar processors. *Technical report*, *Univ. of Wisconsin-Madison*, Nov 1996.

[4] D. S. Henry, B. C. Kuszmaul, G. H. Loh and R. Sami: Circuits for Wide-Window Superscalar Processors. *27th Annual International Symposium on Computer Architecture 2000*, pp. 236-247

[5] M. Butler and Y. N. Patt: An investigation of the performance of various Dynamic Scheduling techniques. In *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pages 1-9, Dec 1992

[6] K. S. Hsiao and C. H. Chen: An Efficient Wakeup Design for Energy Reduction in High-Performance Superscalar Processors. In *proceedings of the 2nd conference on Computing frontiers (CF'05)*

[7] M. Goshima: Research on High speed scheduling logic for out-of-Order superscalar Processor. *Doctoral dissertation, Kyoto University*, March 2004

[8] H. Ichibayashi, R. Shioya, H. Irie, M. Goshima and S. Sakai: Anti-Dualflow Architecture. *Symp. on Advanced Computing Systems & Infrastructures (SACSIS)*, pp. 245—254 (2008).

[9] J. L. Hennessy and D. A. Patterson: Computer Architecture - A quantitative Approach 4th Edition. ISBN – 1-55860-724-2

[10] Virtex-6 FPGA Configurable Logic Block User Guide (UG364)

Acknowledgement

I would like to express my deepest gratitude to all those who have helped me in accomplishment of this research.

I want to thank professor Shuichi Sakai, for giving his valuable suggestions and guidance during this research and specially during the discussion meetings. I would also like to thank him for promptly providing the funds for buying the tools and books etc, which I needed for this research.

I owe my deepest gratitude to my supervisor, associate professor Masahiro Goshima, whose continuous encouragement, precious guidance and support from the beginning, helped me to properly understand various aspects of computer architecture, and in accomplishment of this research. I also thank him for taking out precious time from his schedule to listen about the ideas which came to my mind and guiding me thoroughly about its pros & cons, and showing me the hidden aspects.

I again want to thank him for granting me long duration leaves when I became ill and also when I wanted to go to India for my sister's marriage. I also thank him for writing many recommendation letters for scholarship.

I am very much grateful to my senior Mr. Shioya Ryota. He not just helped me by guiding me and clearing my doubts but also with the actual implementation of this research.

I want to extend my thanks to the office staff Mrs. Harumi Yagihara, Mrs. Tamaki Hasebe and Miss Tomoyo Ise for helping me in various official work and also for arranging vegetarian food specially for me, during Christmas party gatherings etc.

As I am a foreign student and not so good at Japanese language, I had difficulty in having conversation with my lab mates, my supervisor etc. But, everyone here at Sakai-Goshima Lab assisted me by using easier Japanese or using English during the conversation to made me feel comfortable in the lab. I am thankful to all of them.