Abstract

The X4CP32 is a novel coarse RPU runtime-reconfigurable general purpose microprocessor. It consists of 3 programming levels, based on a hierarchic array of easily and quickly reconfigurable entities. It brings a new concept of runtime reconfiguration and programming, which is its main strength. Although it’s effectiveness in heavy arithmetic applications, it is suited for virtually any task an application can demand, presenting as a solid option for a general purpose microprocessor.

1. Introduction

Nowadays, more than ever, we are looking for new alternatives to the conventional microprocessor paradigm. Since the advent of Reconfigurable Architectures a lot of application specific (or a few applications specific) architectures were presented, seeking to replace ASIC’s in embedded systems. However, there are only a few projects with the intuit to design a general purpose microprocessor using the reconfigurable logic[1]. The X4CP32 was designed with this goal, and that’s why it has such a solid multi level programming.

In addition to it’s multi level programming, the X4CP32 has a powerful stream based applications support. It can be configured to act as a systolic array, but with a programming power that multiplies it performance over a systolic-only option. Because of that power, it is suited for DSP and any other stream based application, but nonetheless able to perform any application the user require.

In the next section will be presented the X4CP32 architecture. The internal communication will be shown in section III, followed by a configuration explanation in section IV. Section V presents application examples and results, and finally a conclusion in section VI.

2. Architecture

The X4CP32 architecture is based on cells and grains. A cell is a microprocessor with an instruction set common to most general purpose microprocessors, plus a few configuration specific instructions. The grains are connected in a matrix plus they can send and receive data from distant grains. The architecture also counts with a bus to each row and column, and the Main Control, a higher entity responsible for I/O.

2.1. The RPU

The RPU is the entity responsible for X4CP32’s reconfiguration. It consists of a set of 4 cells, an internal memory (G-MEM), a small communication buffer, 3 connection buses and a control unit. They are connected to their neighbors (top, bottom, left and right) through their cells by 2 half-duplex dedicated bus to each neighbor (as shown in Picture 1), and to all other grains in the same column and row through a vertical and horizontal bus. A RPU is also connected to the Main Control via shared bus.

The RPU has a set of configuration options, thus being able
to act as a processor with a program flow, a set of 4 systolic
cells or a memory block. A better vision on RPU’s
configuration modes will be presented in section IV.

2.2. The Cell

The cell (Picture 2) is the smallest entity in X4CP32, the one
responsible for operations execution and program flow itself.

It consists in a ALU, a by-pass independent circuit to route
data to a neighbor cell, a memory set (C-MEM), some registers,
an internal stack (C-LIFO) and a control unit [3, 4]. A cell has 5
neighbors: 3 in the same RPU plus 2 outside, in different
neighbor grains, and a connection to the RPU memory (G-
MEM). However, it can only receive data from two other cells
and send to a third one at the same time, excluding the routing
system, which can send and receive data from two other cells.

The ALU operates in 32 bits, and it’s capable of executing
sum, subtraction, multiplication and division, both in fixed and
floating point numbers, as well as basic logic operations. It can
also be ordered to shift left or bypass.

C-MEM is a 1024 x 32 bits memory set, capable of one
reading and one writing operation per cycle. It can receive data
from C-LIFO, another cell, ACC or from itself, and can send
data to the ALU operator registers, to another cell or to itself.

C-LIFO is a 64 x 32 bits internal stack, and has almost the
same communication options as the C-MEM.

PL and PR are multiplexers configured to receive data from
one cell each, or from G-MEM. PO sends data to another cell
or to G-MEM.

2.3. The Main Control

The Main Control is, in a last instance, the managing unit. It
reads the program and configuration commands in the main
memory, ordering the grains to perform them later on.

Each RPU has an unique address, so does each cell inside a
RPU, and the Main Control is responsible for providing every
RPU with their instructions and data through a shared bus that
connects all the grains.

2.4 The Data Path

X4CP32’s architecture possesses 3 distinct buses [5]:

- **RPU/RPU bus**: the half-duplex bus connecting a RPU to
  its neighbors. It is, in fact, the same bus used to
  connect neighbor cells. The RPU itself has no exclusive
  communication to its neighbors, only via its cells.

- **Column/Row bus**: Each column and each row of grains
  possesses its own bus. These buses can be used in a
number of ways: the Main-Control can use them to insert RPU configuration in one RPU’s Control Unit using a RPU-Level instruction. One RPU can also use it to communicate with any other RPU despite their distance. This can be done by using another RPU-Level instruction sent from the Main Control to that RPU.

- **Control-Data bus:** This bus connects all RPU’s to the Main Control. The Main Control can send data to any RPU through it. This data can be also a Cell-Level instruction, which will be executed by the RPU when in the Processor Execution Mode. The RPU’s modes will be detailed later on.

### 3. Internal Communication

The communication of it’s components is one X4CP32’s strongest features. Cells can operate and send data at the same time, grains can be reconfigured on the fly, converting instructions received from the Main Control to its cells.

#### 3.1. Instructions

The instructions are the main difference in X4CP32 to other reconfigurable architectures. They are needed in every step taken in any part of the architecture, thus being divided in 2 levels.

1) **The Cell-Level instructions:** are instructions executed for the cells. They are usually sent to a cell’s C-MEM from the Main Control via RPU’s G-MEM, but can also be a translation of a RPU Level instruction or an instruction received from another cell. A total of 32 instructions consist a cell’s instruction set, where 29 of them can be found in most simple microprocessors and consist of arithmetical, logical and jumping instructions, plus 3 special reconfiguration instructions.

2) **The RPU-Level Instructions:** are in a total of 7. These instructions are sent from the Main Control to a RPU, and consist basically in configuration or communication instructions. Some instructions are decoded in the RPU’s control unit and turned into Cell-Level instructions which are sent to the cells, depending on the instruction. Others are executed in the RPU’s control unit.

#### 3.1. Multi Level Programming

The Main Control reads both Cell Level instructions and RPU Level instructions in the main memory. A RPU Level instruction contains the destiny RPU’s address, so the Main Control sends this instruction to the target RPU, which will execute the instruction immediately. This instruction may contain reconfiguration information, which will cause the RPU to reset it’s cells and reconfigure them according to the instruction. In case it’s a Cell Level instruction, the Main Control just sends it to the target RPU’s G-MEM, and it will be responsible for sending this instruction to a cell’s C-MEM, and only in the cell this instruction will be executed, as a part of the cell’s program.

#### 3.2. Routing

Another important feature in X4CP32, cell can route instructions while operates at the same time, with no extra processing cost. All cells have a completely independent datapath connecting it’s ports, as shown in Picture 2. It consists of 1 register and 2 multiplexers, both of them connected to all ports. While in reconfiguration process, a RPU can configure it’s cells to open 2 of it’s port with routing intuit. The entrance routing port can be chosen from any one of the cell’s ports, except for the port already configured to send data. The exit routing port must be chosen among a cell’s unused ports. Once the cell is totally configured, which takes 4 clock cycles, it will start bypassing every data received from the entrance routing port to the exit routing port through the independent datapath. Choosing a routing path for a cell is a completely optional step for a cell’s configuration. If no routing path is chosen, a cell configuration takes 3 clock cycles only.

#### 3.3. Synchronism

Every port of a cell represents a connection to a neighbor cell or to the RPU’s G-MEM. In the middle of this connections
ine registers, one for each port. This 33 bits register stabilizes the data received, or sent, before a cell uses it. In 32 of the register’s bits are kept the data bits, since all data in X4CP32 are in 32 bits. The 33rd one is actually a flag to indicate whether the data contained in that register is valid or not. Every time a cells consumes the data in one of these registers it sets this flag to “invalid”, and it will not operate that data again. Only when the other side of the connection sends a new data this flag is set to “valid” again, and it will not send another data until the consumer cell uses the previous one. This way a simple but useful synchronism is attained, and a cell will never operate and invalid data nor lose data because it took to long to consume the previous data.

4. RPU’s Configurations

This is the actual reconfigurability in X4CP32. The grains can assume 3 different Execution Modes as well as several Connection Modes. An Execution Mode contains information on how a RPU will operate, and how it’s cells will be configured. A Connection Mode contains information on how the cells will be linked, which is the same to say which ports will be active, and whether they will be used for entering or exiting. Connection Modes are only attained when the RPU is in Cell Block Execution Mode, in the other Execution Modes the ports’ configuration change very often. Here lies the difference to classical parallel computing, since the X4CP32 can not only act as a parallel processor, but it can also act as a systolic processor, switching between these modes in runtime.

4.1. Processor

The most complex Execution Mode a RPU can assume. When a RPU is in Processor Execution Mode it acquires independence to execute a program itself. It’s top left cell assumes the processor function (and configuration). This cell is now able to follow a code of Cell Level instructions loaded to it’s C-MEM. This cell is called PC. The other 3 cells in the RPU now receive instructions from the PC cell, which writes the instructions directly into their control logic. These instructions are also Cell Level instructions, and their parameters include the target cell. They are usually arithmetic instructions, but other instructions such as LOAD or STORE are often sent from the PC cell. The other 3 cells are called Dynamic ALU, because it’s operations and communications change constantly.

The PC cell follows the program flows as indicated, usually heading to the next instruction but a JUMP similar instruction can alter this flow. The new instruction is transferred to PC’s control logic for analysis. If the PC cell is the instruction’s target it will be executed and the next instruction will be read. If not, the PC cells sends the instruction to the target cell to proceed to the next instruction. This loop repeats until a reconfigure instruction reaches the RPU. The Processor Execution Mode is the only mode where cell programming is possible, and a good programmer will find the best way to distribute instructions among cells to attain as much parallelism as possible. The PC cell and all Dynamic ALU cells can route and operate ate the same time, as long as the routing restrictions are followed.

4.2. Memory Set

In the Memory Set Execution Mode a RPU assumes a memory function. In case an application requires more memory that it’s available to a RPU, it’s possible to configure a neighbor RPU in Memory Set to use it’s memory. When in this Execution Mode the RPU’s only purpose is to provide extra memory to neighbor grains. It’s G-MEM and all of it’s cells C-MEM becomes available for any other RPU to address and use as demanded. This Execution Mode is not completely detailed yet, thus waiting for later versions of X4CP32 to be implemented.

4.3. Cell Block

Cell Block is the Execution Mode where X4CP32 acts as the usual reconfigurable architectures, operating incoming data in a systolic way. Once a RPU assumes the Cell Block Execution Mode it’s cells are immediately attached to one of the following arithmetical/logical operations: fixed-point sum, subtraction, multiplication or division, floating-point sum, subtraction, multiplication or division, OR, AND, NOR, NAND, XOR, NXOR, which means all the cell’s ALU’s operations set. A cell in the Cell Block Execution Mode is also called Static ALU, for it’s properties won’t change As the operation is defined, the cell also is configured as one of the Connection Modes, defining all this RPU’s cells’ connections to each other and neighbors. After all the configuration, which takes 3 clock cycles, the cells start operating all the data coming from one of it’s entrance ports and sending the result to a neighbor cell through it’s exit port. The ports are configured during the reconfiguring period, and don’t change until another reconfiguration instruction reaches the RPU. Some Connection
Modes need to use a cell’s routing system, in this case a different routing path cannot be chosen. Otherwise, if the routing rules are followed, a cell can operate and route at the same time.

A RPU matrix supports different grains in different Execution Modes at a same time, for the grains are completely independent from each other. So it’s possible to have Processor grains executing a program and sending data to neighbor Cell Block grains, which operate the received data and deliver the results to another Processor RPU or returns to the original Processor RPU itself. Not only possible, but necessary to use X4CP32’s full capacity.

4.4. Connection Modes

Connection Modes are not Execution Modes, but a complement to the Cell Block Execution Mode. Because of its vast array of possible configurations and because of the very definition of Execution Modes, the Cell Block Execution Mode has its own subset of configurations. They mainly define which cells are connected to which, both in the same or in a different, but neighbor, RPU. There is a pre-defined array of possible Configuration Schemes, as shown in Picture 4.

The arrows define an entrance or exit, depending on its direction. A cell can only have 2 entrance ports and 1 exit port at the same time, so any value greater than these mean an obligatory routing path.

However, a Connection Mode does not only consist of the 16 pre-defined Configuration Schemes. For each Configuration Scheme it is possible to perform three 90 degree rotations, as in Picture 5, in a total of 4 different Connection Modes for each Configuration Scheme. This leads us to 64 possible Connection Modes.

5. Applications

This section presents two application examples performed in X4CP32 and compilation results. The first application detailed below is a FIR (Finite Impulse Response) Filter and a resampler calculating new samples. The second application is a 2 dimensions DCT (Discrete Fourier Transforming).

5.1. FIR Filter

This application consists of a resampler and a convolution to implement a FIR Filter [5]. The resampler is implemented in a Processor Execution Mode RPU, and the convolution in a set of 6 Cell Block Execution Mode grains, as seen in Picture 6.

The Processor Execution Mode RPU has a simple code within its PC cell, responsible for receiving samples from elsewhere and, by using interpolation, doubling the sampling rate. These new samples are sent to the RPU’s east neighbor, which starts calculating the convolution.

All cells marked with a “*” are performing float-point multiplication, cells marked with a “+” are performing float-
point sum. The cells which are not marked at all perform no operation, just routing data from incoming ports.

The incoming data from the Processor RPU is both routed and multiplied in the top-left Block Cell RPU. The routed data flows to the right, while the multiplication result is sent to the bottom cell. This cell sums this multiplication result with incoming data from its east neighbor. The result is sent back to the Processor RPU. This same mechanism repeats throughout the RPU’s set, with but one exception. In the top-right and bottom-right grains the communication is done by using vertical bus to send data. All data coming from the top-right RPU’s west neighbor is routed to G-MEM instead of the usual routing to another cell. When in G-MEM, the data is sent to its south neighbor’s G-MEM via vertical bus. Only then this data is sent to the only multiplier cell inside the RPU, continuing the data flow. The opposite way is simpler though, since the no operating cells route the data back to the upper row and the data flow continues.

Once the first data is sent from the Processor RPU to the convolution RPU set, it takes XX cycles for the first result, for instance, the initial latency is XX cycles. Afterwards a new data arrives in YY cycles.

5.2. 2D DCT

The 2D DCT (Two Dimensional Discrete Cosine Transform) is a mathematical function used to transform the space domain representation into the frequency domain representation. It is a part of the JPEG image compression process.

This implementation is based on a dedicated JPEG compressor architecture [6, 7]. The algorithm used for calculating the 2D DCT is described with more details in [6] and in [7]. It is based on separability property and actually calculates two 1D DCT and 8x8 matrix transposition between. The generic 2D DCT is shown in Picture 7.

![Picture 7 – Generic 2D DCT](image)

The algorithm used for calculating the 1D DCT was first suggested in [8], modified by [9] and corrected by [6]. It has 6 steps and 4 constant values and seen in Picture 8.

![Picture 8 – 1D DCT Algorithm](image)

**Step 1:**

\[
E_0 = A_0 + A_7 \\
E_1 = A_1 + A_6 \\
E_2 = A_2 + A_5 \\
E_3 = A_3 + A_4 \\
E_4 = A_2 + A_5 \\
E_5 = A_0 + A_7
\]

**Step 2:**

\[
C_0 = B_0 + B_5 \\
C_1 = B_1 + B_4 \\
C_2 = B_2 + B_6 \\
C_3 = B_1 + B_4 \\
C_4 = B_0 + B_5 \\
C_5 = B_2 + B_6
\]

**Step 3:**

\[
D_0 = C_0 + C_3 \\
D_1 = C_0 + C_3 \\
D_2 = C_2 \\
D_3 = C_1 + C_4 \\
D_4 = C_2 + C_5 \\
D_5 = C_4 \\
D_6 = C_5 \\
D_7 = C_6 \\
D_8 = C_7
\]

**Step 4:**

\[
E_0 = D_0 \\
E_1 = D_1 \\
E_2 = M_3 * D_3 \\
E_3 = M_1 * D_7 \\
E_4 = M_4 * D_0 \\
E_5 = D_5 \\
E_6 = M_1 * D_3 \\
E_7 = M_2 * D_4 \\
E_8 = D_8
\]

**Step 5:**

\[
F_0 = E_0 \\
F_1 = E_1 \\
F_2 = E_6 + E_8 \\
F_3 = E_6 - E_8 \\
F_4 = E_4 + E_8 \\
F_5 = E_5 - E_3
\]

**Step 6:**

\[
S_0 = F_0 \\
S_1 = E_4 + F_7 \\
S_2 = F_2 \\
S_3 = F_5 - F_6 \\
S_4 = F_1 \\
S_5 = F_5 + F_6 \\
S_6 = F_3 \\
S_7 = F_4 - F_7
\]

Where:

\[
M_1 = \cos \left(\frac{4\pi}{16}\right) \\
M_2 = \cos \left(\frac{6\pi}{16}\right) \\
M_3 = \cos \left(\frac{2\pi}{16}\right) \cdot \cos \left(\frac{6\pi}{16}\right) \\
M_4 = \cos \left(\frac{2\pi}{16}\right) + \cos \left(\frac{6\pi}{16}\right)
\]

**Picture 8 – 1D DCT Algorithm**

The algorithm was implemented in two 5x4 matrix of grains, one matrix for the first part and another for the second one, since we are dealing with a 1D algorithm but a 2D problem. This implementation is shown in Picture 9.

The grains in gray are in Processor Execution Mode, while all the others are in Cell Block Execution Mode. Empty arrows are routing paths, fill arrows are usual data results. Neighbor grains communicate through it’s cells neighborhood connections, unless otherwise specified. Vertical/Horizontal bus uses for data sending are in M(XX) notation.

This algorithm requires a heavy amount of data sending to different cells or, in other words, routing. Since a single cell cannot route 2 distinct data at the same time to different destinations, a few of them are in a virtual “routing-only” mode. While it’s not a real configuration, it can be attained when a cell has it’s usual routing path plus it’s ALU is configured for “no operation” with incoming data. This way any incoming data is immediately sent to the cell’s exit port. The usual routing path takes 2 clock cycles, while in “no operation” it takes 3 cycles to completely bypass a cell.

Synchronism is attained through data dependency. By using the pipelining properties it’s possible to define a delay diagram for the application and defining an ideal order and timing for inserting data.

This implementation has a small difference over the original algorithm. The cell’s ALU VHDL current state does not contain float-point operations. Although the architecture is
able to perform such operations, it would be only possible to extract estimated results without ALU’s VHDL description. And since this application requires floating-point operations, all the constant values $M_1$, $M_2$, $M_3$ and $M_4$ were multiplied by 256 so only fixed-point operations are needed. Below are the values used in the implementation.

$M_1$

- Actual value: 0.707107
- Value used: 0.707031

$M_2$

- Actual value: 0.382683
- Value used: 0.378906

$M_3$

- Actual value: 0.541196
- Value used: 0.539062

$M_4$

- Actual value: 1.306563
- Value used: 1.304687

These values produce an average error rate of 0.00196337, the same magnitude of original implementation. The 256 constant was chosen so the average error rate was similar, which turned the comparison between the 2 implementations possible.

However, the multiplication solution has a drawback. All the multiplication results are in a different magnitude from the addition or subtraction results. To solve this problem, new multiplication operations were included in the data flow, hereby represented as "[*]".

But two of the final results, $S_0$ and $S_4$, have no multiplication in their data flow, so the other results need to be in regular magnitude in order to be used. For this reason, the Processor Execution Mode grains have to divide the different-magnitude results, which is done with 8 right shifts, and place them in correct order for the second 1D DCT to use them., which is the exact same function as a transbuffer’s.

$X4CP32$’s 2D DCT has 2 different implementations, one of them focusing on an area economy, the other focusing on higher speed.

The first implementation is attained by re-inserting transposed data from the 1D DCT again into the datapath. It causes an increased delay between 8x8 matrix results, but it has an expressive area economy, for it needs only 20 grains. It only takes a different programming in the Processor Execution Mode grains for this implementation focus to take place, re-inserting data into the datapath instead of sending them for another 1D DCT block.

The second implementation needs twice as many grains as the first, but it has 2 clear advantages over the first. The major advantage is the speed gain, which is desired in most cases, specially when time is critical. The second one, while not so clear at first glance, is it’s easier programming.

The initial latency in both implementations is of 160 clock cycles. In the first one, a new 8x8 matrix is calculated in 160 clock cycles, with a full pipeline. In the second one, a new 8x8 matrix is calculated in 80 clock cycles, which means half of the time. The original implementation [6] has a latency of 163 clock cycles and generates a new 8x8 matrix in 64 clock cycles, which is which means a 25% smaller number of clock cycles than the second implementation.

But when time comparison is taken in account, $X4CP32$’s implementation is 45% more efficient than the application-specific architecture’s implementation, calculating 90.2 640x480 or 35.2 1024x768 images per second in the first implementation. Since the second $X4CP32$’s implementation is twice as fast as the first one and 190% faster than the application-specific architecture’s implementation, it generates 180.4 640x480 or 70.4 1024x768 images per second.

Both architectures were placed in Altera’s FLEX10K family devices [10], but $X4CP32$’s clock of 69 MHz is more than 3 times the application-specific architecture’s clock of 19MHz.
5.3. Compilation Results

The X4CP32’s cell was compiled in Altera’s Quartus II software, obtaining the following results:

<table>
<thead>
<tr>
<th>Device</th>
<th>EPF10K200SBC600-1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Logic Elements</td>
<td>1512/9984 (15%)</td>
</tr>
<tr>
<td>Total Pins</td>
<td>442/470 (94%)</td>
</tr>
<tr>
<td>Total EAB bits</td>
<td>34816/98304 (35%)</td>
</tr>
<tr>
<td>Clock</td>
<td>69.35 MHz</td>
</tr>
</tbody>
</table>

The implemented cell’s ALU still lacks float-point operations, as explained above. This miss had a minor influence in above implementations, but it’s inclusion must not alter the 69.35 MHz clock. The RPU is already in implementation process, and new results will soon be obtained.

6. Conclusion

This paper introduced the X4CP32, a novel general purpose coarse-RPU reconfigurable architecture with 3 programming levels, its reconfiguration, application mapping and compilation results. Its solid performance is very suited for DSP applications, or any other depending heavily on arithmetic operations. But its configuration and programming power makes it useful for virtually any application. In an application example it performance was up to 3 times better than an ASIC implemented in the same conditions.

The future works include finishing the cell’s and RPU’s implementation, which must happen in a few months. A simulator and assembler [11, 12, 13] project is also a future target, enabling the testing of applications in a significantly lower amount of time.

7. References

[10] www.altera.com