Custom Processor Design Using NISC: A Case-Study on DCT algorithm

Bita Gorjiara, Daniel Gajski
Center for Embedded System Computers, University of California Irvine
{bgorjiar, gajski}@ cecs.uci.edu

Abstract

Designing Application-Specific Instruction-set Processors (ASIPs) usually requires designing a custom datapath, and modifying instruction-set, instruction decoder, and compiler. A new alternative to ASIPs is No-Instruction-Set-Computers (NISCs) that eliminate the instruction abstraction by compiling programs directly to a given datapath. The compiler analyzes the datapath and extracts possible operations and data flows. The NISC approach simplifies and accelerates the task of custom processor design. In this paper, we present a case-study of designing a custom datapath for a 2-D DCT algorithm. We applied several optimization techniques such as software transformations, operation chaining, datapath pipelining, controller pipelining, and functional unit customization to improve the quality of the design. Most of the techniques are general and can be applied to other applications. The result of synthesizing our final custom datapath on a Xilinx FPGA shows 7.14 times performance improvement, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation.

1. Introduction

In general, custom hardware designs are several orders of magnitude faster than the equivalent software implementation. However this performance efficiency is achieved at the expense of lower productivity and higher design cost. To bridge the productivity and performance gap, many design methodologies have been proposed. The Application-Specific Instruction-set Processor (ASIP) design is one of the promising approaches [2]. ASIPs improve the performance of an application by running it on a customized datapath, using custom instructions. ASIP designers usually prefer to focus on the datapath design, however they have to spend substantial amount of time on modifying the instruction-set and the controller (instruction decoder). Designing the custom instructions can be difficult and complex. The custom instructions should represent the capabilities of the corresponding datapath. The designer should also consider the effects of the custom instructions on both the instruction decoder and the compiler. Such constrains not only complicate and slow down the design process, but also impose unnecessary limitations on the possible datapath customizations.

A new alternative to ASIP is No-Instruction-Set-Computer (NISC) [3]. Similar to horizontally micro-coded architectures, a NISC compiler generates code to control the datapath at every clock cycle. However, instead of using any abstraction such as instruction-set or microcode, the NISC compiler directly generates the control signal values of every component in the datapath for every clock cycle. A NISC designer needs to only focus on designing the datapath, i.e. selecting the components and connecting them together. There is no need for designing instruction-set and instruction decoder, or updating the compiler. The NISC compiler inputs the datapath as a netlist of RTL components, and automatically analyzes and extracts possible operations. The datapath netlist contains components such as bus, multiplexer, register, register-file, memory, and functional unit. For each component, the functionalities are defined and linked to proper values of component's control signals. After compiling the program onto the given datapath, the compiler generates a string of control values, called Control Word (CW), for each cycle. These control words are stored in a control memory and are applied to the datapath by the controller at every cycle. If the size of the program is small enough, then it is also possible to generate the control words via logic or ROM.

In this paper, we present a case-study of using NISC approach for designing a custom architecture for a Discrete Cosine Transform (DCT) algorithm. We start from a general purpose datapath similar to MIPS and iteratively customize the datapath to achieve significant improvement in terms of power, performance and area. The exploration methodology is general and can be applied to other multimedia algorithms. Taking the advantage of the NISC design methodology and the compiler, we were able to design and explore more than 10 different architectures over a course of one week. The Verilog files of all the designs presented in this paper are available at [8]. Our results show 7.14 times performance improvement, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation.

The rest of the paper is organized as follows: Section 2 presents an overview of the NISC design methodology. Section 3 explains applying NISC approach to design of a custom DCT architecture. Section 4 compares different design points in terms of performance, power, energy, and area. Section 5 compares our best NISC-based DCT implementation to a commercial manual design.

2. Overview of NISC approach

A NISC is composed of a pipelined datapath and a pipelined controller that drives the control signals of the datapath components at each clock cycle. The controller has a fixed template and is usually composed of a Program Counter (PC) register, an Address Generator (AG) and a Control Memory (CMem). The control values are stored in a control memory. For small size programs, the control values can also be generated via logic in the controller. The datapath of NISC can be simple or as complex as datapath of a processor. Figure 1 shows a sample NISC architecture with a memory-based controller and a pipelined datapath that has partial data forwarding, multi-cycle and pipelined units.

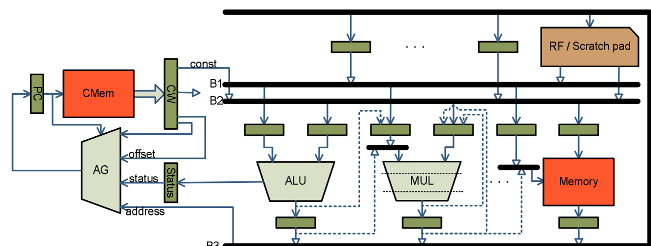

Figure 1- A sample NISC architecture.

Figure 2 shows a NISC-based design flow for implementing an application on a custom hardware. In NISC, the datapath can be generated (allocated) using different techniques. For example, it can

be an IP, reused form the previous designs, generated by High-Level Synthesis, or directly specified by a designer. In our current implementation, we use an XML (eXtensible Markup Language) file to capture the netlist of components in the datapath. A component can be a register, register-file, tri-state buffer, multiplexer, functional unit, memory, or bus. The functionalities of components are linked to the timing information of their control values. The program, written in a high-level language such as C, is first compiled and optimized by a front-end and then mapped (scheduled and bound) on the given datapath. The compiler generates the control words as well as the contents of data memory. The generated results and datapath information are translated to a synthesizable RTL design, described in Verilog, that is used for simulation (validation) and synthesis (implementation). After synthesis and Placement and Routing (PAR), the accurate timing, power, and area information can be extracted and used for further datapath *refinement*. For example, the user may add functional units and pipeline registers, or change the bit-width of the components and observe the effect of modifications on precision of the computations, number of cycles, clock period, power, and area. In NISC, there is no need to design the instruction-set because the compiler automatically analyzes the datapath and extracts possible operations and branch delay. Therefore, the designer can refine the design very fast.

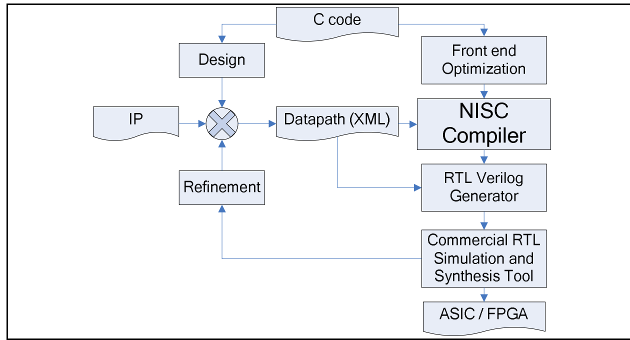

Figure 2- NISC based design flow.

To give the designer more control over the datapath and application mapping, the compiler also allows pre-binding of variables and operations. The unbound variables and operations are mapped by the compiler automatically. If an application has a particular operation that is not supported by a given datapath, then the compiler raises an exception and terminates the compilation.

3. Case study: DCT implementation

In this section, a short introduction on DCT is presented, and then a custom datapath for the DCT is designed and refined using NISC methodology. The Discrete Cosine Transform (DCT) [1] and Inverse Discrete Cosine Transform (IDCT) are important parts of JPEG [4] and MPEG [5] standards. MPEG encoders use both DCT and IDCT, whereas MPEG decoders only use IDCT. The definition of DCT for a 2-D 8×8 matrix of pixels is as follows:

$$F[u,v] = \frac{1}{N^2}\sum_{m=0}^{N-1}\sum_{n=0}^{N-1} f[m,n]\cos\frac{(2m+1)u\pi}{2N}\cos\frac{(2n+1)v\pi}{2N}$$

Where u, v are discrete frequency variables ($0 \le u, v \le 7$), $f[i, j]$ gray level of pixel at position (i, j), and $F[u,v]$ coefficients of point (u, v) in spatial frequency. Assuming N=8, matrix C is defined as follows:

$$C[u][n] = \frac{1}{8}\cos\frac{(2n+1)u\pi}{16}$$

Based on matrix C, an integer matrix $C1$ is defined as follows:

$$C1 = \text{round}(\mathit{factor} \times C)$$

The $C1$ matrix is used in calculation of DCT and IDCT:

$$F = C1 \times f \times C2$$

where, $C2 = C1^T$. As a result, DCT can be calculated using two consecutive matrix multiplications.

3.1 Implementing DCT using general-purpose datapaths

Figure 3 shows the C code for multiplying two given matrix A and B using three nested loops. Using a MIPS M4K ™ Core processor [6], the matrix-multiplication-based DCT takes 13058 cycles to compute [3]. However, given the MIPS datapath, the NISC implementation takes 10772 cycles. The 20% reduction in number of cycles is because of the finer-grained control that NISC compiler has over the datapath compared to traditional compilers that use instruction-set abstraction. We generated the synthesizable hardware description for our NISC-style MIPS (NMIPS), and synthesized it using Xilinx ISE 6.3. In our implementation, the bus-width of the datapath is 16-bit, and it does not have any integer divider or floating point unit. The clock frequency of 78.3MHz was achieved after synthesis and Placement-and-Routing.

All the experiments in this section are synthesized on Xilinx FPGA package Virtex2V250-6 using Xilinx ISE 6.3 tool. Two synthesis optimizations of retiming and buffer-to-multiplexer conversions are applied during optimization to improve the performance. In these experiments, we set the PAR effort to the highest level possible for maximum clock speed.

```
for(int i=0; i<8; i++)
  for(int j=0; j<8; j++){
    sum=0;
    for(int k=0; k<8; k++)
      sum = sum + A[i][k] ×B[k][j];
    C[i][j] = sum;
  }
```

Figure 3. C-Code of matrix multiplication

Figure 4 shows a simple general-purpose datapath (GPD) that includes an ALU, a Register File (RF), a multiplier (Mul), a data memory (Mem), a Comparator (Comp), and three buses. The RF has 32 registers, and ALU and Comp are designed to execute various C operations listed in Table 1. In NISC, the controller has a fixed structure and includes an Address Generator (AG), a Program Counter (PC), and a Control Memory (CMem). To support function call, a Link Register (LR) is added to the controller. The control and data memories are implemented using FPGA Block RAMs. The Block RAMs are synchronous and need to be driven by the clock. On each FPGA package, 24 Block RAMs exist where each has 16-Kb capacity. In our experiments, only two Block RAMs per architecture are used. Also, the FPGA package has 24 pre-synthesized multipliers, which only one is used.

Component	Operations
ALU	Add, Sub, And, Or, Xor, Shift-right, Shift-left, Shift-right-unsigned, Negate, Not
Comparator	Equal, Not-equal, Greater-or-equal, Greater-than, Less-than, Less-or-equal, Greater-or-equal-unsigned, Greater-than-unsigned, Less-than-unsigned, Less-or-equal-unsigned

Table 1. ALU and Comparator operations

To support constant-based operations and jumps, a 10-bit constant and a 10-bit offset is added. The total number of bits in a single control word, including the constant and the offset bits, is 61. The NISC compiler generates about 50 control words. The clock frequency of GPD is 92.6 MHz.

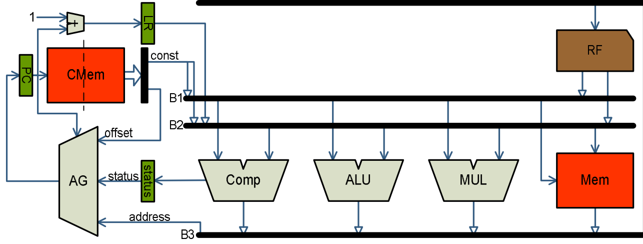

Figure 4. Block diagram of GPD

In the rest of this section, we use different techniques to improve performance, area and power of the design. The techniques include:

1- Software transformations: unrolling the matrix multiplication loops to increase the parallelism in the code, and applying simple code transformations to reduce costly operations.

2- Using Multiply-and-Accumulate (MAC) unit: this technique improves the performance by chaining the two operations without accessing the Register File.

3- Adding pipeline registers to the datapath: if applied properly, this technique decreases the overall delay by reducing the clock period and increasing parallelism. Additionally, the power consumption decreases due to the reduction in switching activity.

4- Adding pipeline registers to the controller: although this technique increases branch delay (and hence the total number of cycles), the controller pipelining can help in reducing the critical path

5- Removing unused parts of ALU, comparator and register file: in a general-purpose datapath, all the operations supported in C, must be handled by the datapath. However, in a customized datapath, only the operations used by a specific application are supported. This optimization improves the area and performance.

6- Reducing the bit-width of some components without affecting the precision of the DCT calculations: this optimization reduces the area.

3.2 Designing a custom hardware for DCT

In general, customization of design involves both software and hardware transformations. In this section, we first apply the software transformations, and then customize and refine the datapath accordingly. Currently, the transformations are applied manually. In future, they can be applied automatically by tools.

3.2.1 Software transformations

To increase the parallelism, we unroll the inner-most loop of the matrix multiplication code. The transformed code is shown in Figure 5. Note that operation "*" represents accessing the value of a pointer (i.e. loading from memory). Next, we apply other software transformations to reduce the costly operations: To decrease the number of multiplications, we replace $i \times 8$ with i<<3 (i shift left three times). Additionally, to calculate the address, we need two consecutive additions, which may require two chained adders. However, if we replace one of the additions with an OR operation, then we can chain one adder with an OR unit, which is less costly than an extra adder. The conversion is possible in this particular application because of the special values of the constants. For example, *i*8+*const* is equal to *i*8|*const*, because $0 \le const \le 7$ at all time and the first three bits of *i*8 is always zero. Additionally, the two *for* loops can be merged to one, by combining the loops' counters. The new counter is represented by variable *ij*. Figure 6 shows the transformed code after the above modifications.

```
for(int i=0; i<8; i++)
   for(int j=0; j<8; j++){
      i8 = i × 8;
      sum = *(A + i8) × *(B + j);
      sum += *(A + i8 + 1) × *(B + 8 + j);
      sum += *(A + i8 + 2) × *(B + 16 + j);
      sum += *(A + i8 + 3) × *(B + 24 + j);
      sum += *(A + i8 + 4) × *(B + 32 + j);
      sum += *(A + i8 + 5) × *(B + 40 + j);
      sum += *(A + i8 + 6) × *(B + 48 + j);
      sum += *(A + i8 + 7) × *(B + 56 + j);
      C[i][j] = sum;
   }
```

Figure 5. C-code of unrolled matrix multiplication

```
ij=0;
do {
   i8 = ij & 0xF8;
   j = ij & 0x7;
   aL = *(A+ (i8|0) ); bL = *(B + (0|j) );  sum =  aL × bL;
   aL = *(A+ (i8|1) ); bL = *(B + (8|j) );  sum += aL × bL;
   aL = *(A+ (i8|2) ); bL = *(B + (16|j) ); sum += aL × bL;
   aL = *(A+ (i8|3) ); bL = *(B + (24|j) ); sum += aL × bL;
   aL = *(A+ (i8|4) ); bL = *(B + (32|j) ); sum += aL × bL;
   aL = *(A+ (i8|5) ); bL = *(B + (40|j) ); sum += aL × bL;
   aL = *(A+ (i8|6) ); bL = *(B + (48|j) ); sum += aL × bL;
   aL = *(A+ (i8|7) ); bL = *(B + (56|j) ); *(C + ij) = sum + (aL × bL);
   ++ij;
} while(ij!=64);
```

Figure 6. Transformed matrix multiplication C-code

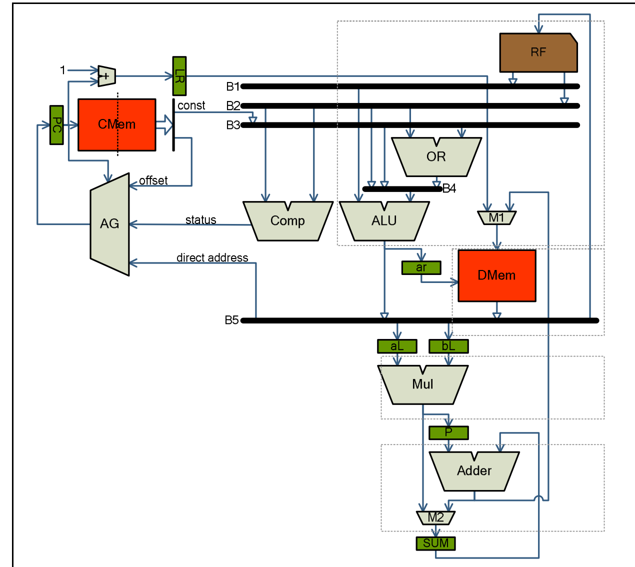

Figure 7. Block diagram of CDCT1

3.2.2 Initial Custom datapath: CDCT1

By looking at the body of loop in Figure 6, four steps of computation can be identified:

1- Calculation of the memory addresses of the relevant elements
2- Loading the values of those elements from data memory,
3- Multiplying the two values,
4- Accumulating the multiplication results .

We design our custom datapath in a way that each of these steps is a pipeline stage. Figure 7 shows the proposed custom pipelined

datapath (CDCT1). The datapath includes four major pipeline stages that are marked in the figure. We have used operation chaining to reduce RF file accesses and decrease register pressure. Chaining the operations improves the energy consumption and performance. The OR and ALU are chained, as well as the Mul and Adder. Note that the chaining of multiply and add forms a MAC unit in the datapath. To assure proper usage of the MAC unit, we enforce mapping the *aL*, *bL*, and *sum* variables, to aL, bL and SUM registers in the datapath. After compilation, the total number of cycles of the DCT is 3080, and the maximum clock frequency is 85.7MHz.

Component	CMem+CW	RF+RF_o	ALU+ALU_o	RF setuptime
Delay (ns)	3.28	2.39	5.4	0.58

Table 2. Critical-path delay breakdown of CDCT1

Table 2 shows the critical-path breakdown of CDCT1. Each column in the table shows the sum of a component delay and its output-interconnect delay. The critical path goes through CMem, RF, B2, B4, ALU, B5, and back to RF.

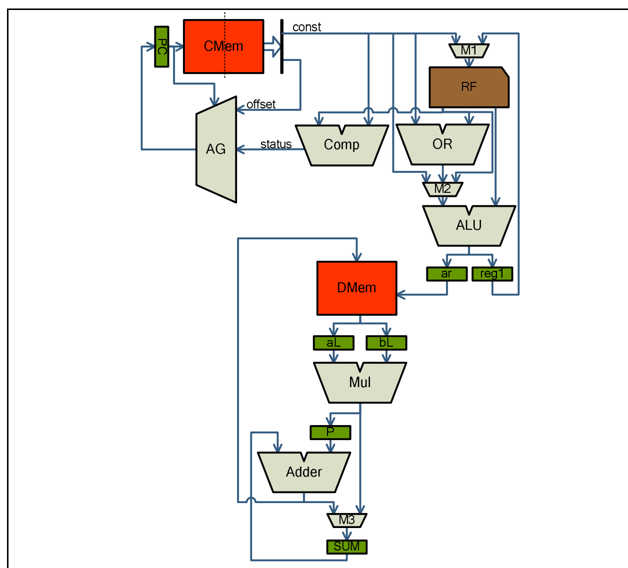

Figure 8. Block diagram of CDCT2

3.2.3 *CDCT2: Bus customization and adding a pipeline register to the datapath*

According to Table 2, ALU and the wire that connects ALU to RF are in the critical path. To reduce the critical path delay, we insert an additional pipeline register (i.e. *reg*1) in the output of the ALU, and call the new design CDCT2 (Figure 8). We also replace all the global buses, including B5, with point-to-point connections. Only the connections that are used by the DCT application are kept. Since there is no function call in DCT, the LR register can be removed. The NISC compiler automatically analyzes the new datapath and regenerates the control words to correctly handle the flow of the data. CDCT2 runs the DCT algorithm in 2952 cycles at the maximum clock frequency of 90MHz. The reduction in number of cycles is due to additional parallelism created by the separation of interconnects. Table 3 shows the breakdown of the critical path of CDCT2. Note that, in CDCT2, the critical path goes through the comparator instead of the ALU. In general, adding pipeline registers combined with retiming optimization is more effective; because, retiming balances the delay of the pipeline stages by moving some of the logic across the pipeline registers. In all the experiments here, we enabled retiming optimization to improve the clock frequency.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	2.93	2.45	3.726	2.06

Table 3. Critical-path delay breakdown of CDCT2

3.2.4 *CDCT3: Eliminating the unused parts of ALU, comparator and RF*

Next, we customize the ALU and comparator for the DCT application. In Figure 6, only Add, And, Multiply and Not-equal (!=) operations are used. The first two operations are executed by ALU, the third by Mul, and the last by Comp. We can simplify the ALU and comparator by eliminating the unused operations. NISC compiler allocates and uses nine registers in RF. Therefore, we reduce number of registers in RF from 32 to 16. The new architecture (CDCT3) runs much faster at the clock frequency of 114.4MHz. The breakdown of critical path delay (Table 4) shows a considerable reduction in the delay of the comparator. Also, the number of fanouts of RF output wires is reduced, and hence its interconnect delay is reduced. These modifications, also, reduce the area significantly.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	2.76	1.64	2.29	2.06

Table 4. Critical-path delay breakdown of CDCT3

3.2.5 *CDCT4 and CDCT5: Controller pipelining*

Looking at the critical paths of the architectures, it is evident that the controller contributes to a major amount of the delay. The CMem, CW, and Address Generator (AG) delays are part of the critical path of CDCT3. To reduce the effect of the controller delay, we insert one pipeline register (i.e. CW register) in front of the CMem. The new architecture (CDCT4) can run much faster at the clock frequency of 147MHz. Table 5 shows a reduction in the critical path delay. On the downside however, the number of cycles of DCT increases to 3080 because of an extra branch delay cycle. Note that the NISC compiler automatically analyzes the datapath and notices the extra branch delay. So, the user does not need to change the compiler manually.

Component	CMem+CW	RF+RF_o	Comp+comp_o	AG+PC setup
Delay (ns)	1.39	1.6	1.74	2.06

Table 5. Critical-path delay breakdown of CDCT4

To further reduce the effect of controller's delay on the clock cycle, we insert another pipeline register (called status register) at the output of the Comp. This register eliminates the AG's delay from the critical path. Table 6 shows the breakdown of the critical path delay of the new architecture (CDCT5). In CDCT5, the critical path goes through the multiplier. Since the multiplier is a pre-synthesized unit in the FPGA package, it is not possible to reduce the critical path delay any further. Note that, CDCT5 has a branch delay of two and runs at the clock frequency of 170MHz. The total number of cycles of DCT has increased to 3208. CDCT4 and CDCT5 occupy larger area than CDCT3 due to the additional CW and status registers.

Component	bL+bL-o	Mul+Mul-o	P setuptime
Delay (ns)	1.29	4.25	0.3

Table 6. Critical-path delay breakdown of CDCT5

3.2.6 *CDCT6: bit-width reduction*

In the final optimization, we reduce the bit-width of some of the components without affecting the precision of the calculations. The goal of this optimization is further reducing the area. We observed that the address-calculation pipeline stage does not need the 16-bit operations. In fact, all the address values are in the range of 0 to 255. Therefore, the bit width of RF, OR, ALU, and Comp are reduced to 8 bits. In this case, the clock frequency remains fixed at 170MHz. Figure 9 shows final design (CDCT6) after all the transformations.

Figure 9. Block diagram of CDCT6

4. Comparing performance, power, energy and area of the NISCs

Table 7 summarizes all the experiments in Section 3. The second column briefly describes the experiments, and the third column shows the bit-width of Control Words. In these experiments, we first mapped DCT to two general-purpose datapaths (NMIPS and GPD). Then, we designed a custom pipelined datapath for DCT called CDCT1. Next, we added an additional pipeline register to CDCT1, simplified the functional units, and added controller pipelining. Finally, we optimized the bit-width of address-calculation pipeline stage and generated CDCT6.

	General Description	CW bit width
NMIPS	NISC with MIPS datapath	76
GPD	A general-purpose NISC architecture	61
CDCT1	Custom NISC for DCT	59
CDCT2	CDCT1 + additional pipeline register + bus transformation	60
CDCT3	CDCT2 with a simplified ALU, comparator and RF	50
CDCT4	CDCT3 + CW register	50
CDCT5	CDCT4 + status register	51
CDCT6	CDCT5 with a 8-bit-width address calculation pipeline stage	51

Table 7. Summary of the experiments

Table 8 compares the performance, power, energy, and area of the all NISC implementations. We synthesized all the NISC architectures on FPGA. After placement and routing and based on the critical path delays, we extracted the maximum clock frequency of each design (shown in the third column).

In Table 8, column fourth shows the total execution time of the DCT algorithm calculated based on number of cycles and the clock frequency. Note that although in some cases (such as CDCT4 and CDCT5) the number of cycles increases, the clock frequency improvement compensates for that. As a result, the total execution delay maintains a decreasing trend.

Column fifth shows the average power consumption of the NISC architectures while running the DCT algorithm. All the designs are stimulated with the same data values. We used Post-Placement and Routing simulation to collect the signal activities, and computed the power consumption using Xilinx XPower tool. Figure 10 shows the power breakdown of different designs in terms of the clock, logic and interconnect power. Column sixth shows the total energy consumption calculated by multiplying power and execution time.

	No. of cycles	Clock freq	DCT exec. time(us)	Power (mW)	Enegy (uJ)	Normalized area
NMIPS	10772	78.3	137.57	177.33	24.40	1.00
GPD	11764	79.5	147.97	150.33	22.24	1.00
CDCT1	3080	85.7	35.94	120.52	4.33	0.81
CDCT2	2952	90.0	32.80	111.27	3.65	0.71
CDCT3	2952	114.4	25.80	82.82	2.14	0.40
CDCT4	3080	147.0	20.95	125.00	2.62	0.46
CDCT5	3208	169.5	18.93	106.00	2.01	0.43
CDCT6	3208	171.5	18.71	104.00	1.95	0.34

Table 8. Performance, power, energy, and area of the DCT implementations

In these experiments, GPD consumes lower power than NMIPS because it does not have any forwarding path. Also, CDCT1 consumes less power than GDP because CDCT1 controls the activation of multiplier by aL and bL registers, while GDP wastes power by always activating ALU, Mul and Comp simultaneously.

CDCT2 consumes less power compared to CDCT1 because of the replacing shared bus B5 with short point-to-point connections. Instead of having a B5 with two fanins and four fanouts, three point-to-point connections are used. This optimization reduces the total bus capacitance and hence, the total power consumption. The diagram of Figure 10 confirms the reduction in interconnect power consumption of CDCT2.

Figure 10. Power breakdown of the DCT implementations

Power consumption of CDCT3 is lower than CDCT2 because of the elimination of unused operations in ALU and comparator. Elimination of operations reduces number on fan-outs of the RF output wires. Therefore, reduction in interconnect power, as well as logic power is achieved. The power breakdown of CDCT3 confirms this fact. Note that as the clock frequency goes up, the clock power gradually increases.

In CDCT4, the power consumption further increases, because of: (1) the higher clock power due to higher clock frequency and higher number of pipeline registers; (2) the higher logic power due to CW register gates; and more importantly, because of (3) the power consumption of logic and interconnects added by retiming algorithm. Since the difference between the delays of the two pipeline stages located before and after CW register is high, the retiming works aggressively to balance the delay. As a result it adds extra logic to the circuit.

In CDCT5, we added the status register to the output of Comp and reduced the critical path. In this case, the retiming algorithm works less aggressive because the delays of the pipeline stages are less imbalanced. As a result, we observe a reduction in logic and interconnect power. The last column of Table 8 shows the normalized area of different designs calculated based on the number of FPGA slices that each design (including memories) occupies. The area trend also confirms the increase in area in CDCT4 followed by a decrease in CDCT5, which we believe is because of the retiming.

Figure 11 shows the performance, power, energy and area of the designs normalized against NMIPS. The total execution delay of DCT algorithm has a decreasing trend except for the GPD that takes many cycles to finish the execution. The power consumption decreases up to CDCT3 and then increases. The energy consumption significantly drops at CDCT1, because of the reduction in number of cycles and power consumption. From CDCT1 to CDCT6, the energy decreases gradually in a slow paste.

As shown in Figure 11, CDCT6 is the best design in terms of delay, energy consumption and area. However, CDCT3 is the best in terms of power consumption. As a result, CDCT3 and CDCT6 are considered the pareto-optimial solutions. Compared to NMIPS, CDCT6 runs 7.14 times faster, consumes 1.69 times less power and 12.51 times less energy. Also CDCT6 occupies 3 times less area than NMIPS. Note that performance of NMIPS is 20% better than performance of a MIPS core. Also, since NMIPS does not have instruction decoder, its area is less than MIPS. In our experiments, we compared the results to NMIPS which is conservative relative to MIPS core. The Verilog description of all the experiments can be downloaded from [8].

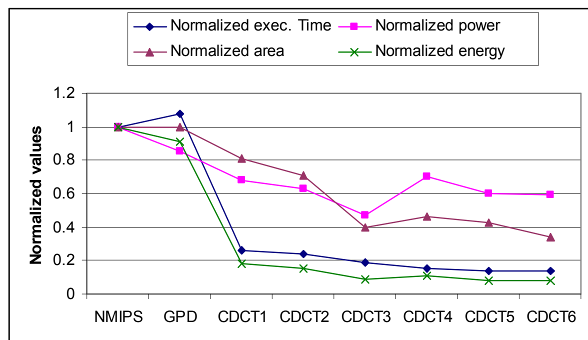

Figure 11. Comparing different DCT implementations

The experiments show that using NISC methodology, a designer (or a tool) can start from a simple general-purpose datapath and iteratively refine and customize it for one or more applications. If properly designed, the custom hardware can be shared by several applications. It is also possible to trade some of the customizations for post-implementation re-programming by maintaining some of the general-purpose features of the initial design.

5. Comparison with a manual design

We have also compared the quality of our final design (CDCT6) with a commercial manual design [7]. In [7], the quality of a manual design after mapping to Xilinx Virtex2V250-6 package is reported. We also used the same package in our experiments to enable the comparison. Their design takes 82 cycles to compute an 8×8 DCT with a 15-bit precision (ours has a 16-bit precision). They have achieved maximum clock frequency of 74MHz on the FPGA package (we achieved 170MHz). Therefore, their total execution time of an 8×8 DCT is 1.1us. Compared to NMIPS that takes 137.57us, the manual design is 125 times faster. This clearly shows two orders of magnitude performance gap between the manual design and software implementation. Compared to CDCT6 that takes 18.71us to compute DCT, the manual design is 17 times faster. These results show that a custom NISC architecture can serve as an intermediate point between software and hardware implementations.

On the other hand, the total area of the manual design is 1365 FPGA slices, while the area of CDCT6 is 169 slices. Note that the low area of CDCT6 allows fitting eight of CDCT6 in the same area as of the manual hardware design. Since the DCT algorithm can usually run on different parts of an image in parallel, the performance of eight CDCT6 is almost eight times of the performance of one. This makes the CDCT6 only two times slower than the manual hardware design. Figure 12 compares the performance and normalized area of the two designs (power is not reported in [7]). Note that it took us about one week to explore different design alternatives while it usually takes significantly longer time to implement and verify a manual designs.

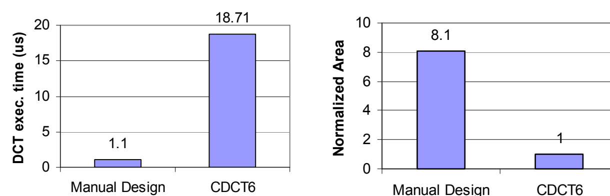

Figure 12. Comparing CDCT6 with a commercial manual design [7]

6. Conclusions and future works

In this paper, we presented a case-study of designing a custom datapath for DCT application using NISC. We started from a general-purpose pipelined datapath and iteratively refined it to achieve better performance, power and area. Our results show 7.14 times performance improvement, 1.64 times power reduction, 12.5 times energy savings, and more than 3 times area reduction compared to a soft-core MIPS implementation. We also compare the quality of our designs to a state-of-the-art commercial manual design. Future works includes simultaneous optimization of a datapath for multiple applications.

Acknowledgements

This work is in part supported by SRC contact 1118.001. We also acknowledge Mehrdad Reshadi for providing his NISC compiler.

References

[1] N. Ahmed, T. Natarajan, and K.R. Rao, Discrete Cosine Transform, *IEEE Trans. On Computers*, vol. C- 23, 1974.

[2] M.K. Jain, M. Balakrishnan, and A. Kumar, ASIP Design Methodologies: Survey and Issues, *In Proc. of International Conference on VLSI Design*, 2001.

[3] M. Reshadi, D. Gajski, An Algorithm for Compiling Programs to Custom Pipelined Datapaths, *In Proc. International Symposium on System Synthesis* (*ISSS05*), 2005.

[4] ISO/IEC JTC1 CD 10918. Digital Compression and Coding of Continuous-tone Still Images - part 1, requirements and guidelines, ISO, 1993 (JPEG)

[5] ISO/IEC JTC1 CD 13818. Generic Coding of Moving Pictures and Associated Audio: Video, ISO, 1994 (MPEG-2 standard)

[6] MIPS32® M4K™ Core, http://www.mips.com

[7] http://www.cast-inc.com/cores/dct/cast_dct-x.pdf

[8] http://newport.eecs.uci.edu/~bgorjiar/projects/NISC/customDCT/