

Mapping Applications to the RaPiD Configurable Architecture*

Carl Ebeling, Darren C. Cronquist, Paul Franklin, Jason Secosky, and Stefan G. Berg

Department of Computer Science and Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350

Abstract

The goal of the RaPiD (Reconfigurable Pipelined Datapath) architecture is to provide high performance configurable computing for a range of computationally-intensive applications that demand special-purpose hardware. This is accomplished by mapping the computation into a deep pipeline using a configurable array of coarse-grained computational units. A key feature of RaPiD is the combination of static and dynamic control. While the underlying computational pipelines are configured statically, a limited amount of dynamic control is provided which greatly increases the range and capability of applications that can be mapped to RaPiD. This paper illustrates this mapping and configuration for several important applications including a FIR filter, 2-D DCT, motion estimation, and parametric curve generation; it also shows how static and dynamic control are used to perform complex computations.

1 Introduction

Field-programmable custom computing machines have attracted a lot of attention recently because of their promise to deliver the high performance provided by special purpose hardware along with the flexibility of general purpose processors. Unfortunately, the promise of configurable computing has yet to be realized in spite of some very successful examples [1, 9]. There are two main reasons for this.

First, configurable computing platforms are currently implemented using commercial FPGAs. These FPGAs are necessarily very fine-grained so they can be used to implement arbitrary circuits, but the overhead of this generality exacts a high price in density and performance. Compared to general purpose processors (including digital signal processors), which use highly optimized functional units that operate in bit-parallel fashion on long data words, FPGAs are somewhat inefficient for performing logical operations and

even worse for ordinary arithmetic. FPGA-based computing has the advantage only on complex bit-oriented computations like count-ones, find-first-one, or complicated bit-level masking and filtering.

Second, programming an FPGA-based configurable computer is akin to designing an ASIC. The programmer either uses synthesis tools that deliver poor density and performance or designs the circuit manually, which requires both intimate knowledge of the FPGA architecture and substantial design time. Neither alternative is attractive, particularly for simple computations that can be described in a few lines of C code.

Our response to these two problems is RaPiD, a coarse-grained configurable architecture for constructing deep computational pipelines. RaPiD is aimed at regular, computation-intensive tasks like those found in digital signal processing (DSP). RaPiD provides a large number of ALUs, multipliers, registers and memory modules that can be configured into the appropriate pipelined datapath. The datapaths constructed in RaPiD are linear arrays of functional units communicating in mostly nearest-neighbor fashion. Systolic algorithms [4], for example, map very well into RaPiD datapaths, allowing us to take advantage of the considerable research on compiling computations to systolic arrays [5, 7]. However, RaPiD is not limited to implementing systolic algorithms; a pipeline can be constructed which comprises different computations at different stages and at different times.

We begin with an overview of the RaPiD architecture; for a more detailed description see [3]. We then give a general description of how computations map to RaPiD using a FIR filter as an example, and then present how the architectural features of RaPiD are used to perform more complex computations found in 2-D DCT, motion estimation, and parametric curve generation.

2 The RaPiD Datapath Architecture

RaPiD is a linear array of functional units which is configured to form a mostly linear computational pipeline. This array of functional units is divided into identical cells which are replicated to form a complete array. Figure 1 shows the cell used in RaPiD-1, the

*This work was supported in part by the Defense Advanced Research Projects Agency under Contract DAAH04-94-G0272. D. Cronquist was supported in part by an IBM fellowship. P. Franklin was supported by an NSF fellowship.

first version of the RaPiD architecture. This cell comprises an integer multiplier, three integer ALUs, six general-purpose “datapath registers” and three small local memories. A typical single-chip RaPiD array would contain between 8 and 32 of these cells.

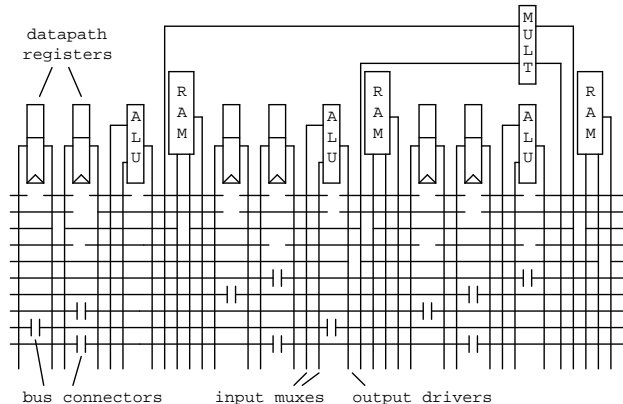


Figure 1: A basic RaPiD cell which is replicated left to right to form a complete chip. RaPiD-1 contains 16 cells similar to this one, with fourteen 16-bit buses.

2.1 Datapath Composition

The functional units are interconnected using a set of segmented buses that run the length of the datapath. The functional units use a $n : 1$ multiplexer to select their data inputs from one of the $n - 2$ bus segments in the adjacent tracks. The additional inputs provide fixed zero or feedback lines, which can be used to clear and hold register values, or to use an ALU as an accumulator. Each functional unit output includes optional registers to accommodate pipeline delays and a set of tristate drivers to drive their output onto one or more bus segments.

The buses in different tracks are segmented into different lengths to make the most efficient use of the connection resources. In some tracks, adjacent bus segments can be connected together by a bus connector as shown in Figure 1. This connection can be programmed in either direction via a unidirectional buffer or pipelined with up to three register delays, allowing data pipelines to be built in the bus structure itself.

RaPiD’s ALUs perform the usual logical and arithmetic operations on one word of data. The ALUs can be chained for wide-integer operations. The multiplier inputs two single-word numbers and produces a double-word result, shifted by a statically programmed amount to maintain a given fixed-point representation. Both words of the result are available as separate outputs.

The datapath registers serve a variety of purposes in RaPiD. These registers can be used to store constants loaded during initialization and temporary val-

ues. They can be used as additional multiplexers to simplify control; like any functional unit, the registers can be disabled. They are also used while routing RaPiD applications to connect bus segments in different tracks and/or for additional pipeline delays.

In many applications, the data is partitioned into blocks which are loaded once, saved locally, reused as needed, and then discarded. The local memories provided in each cell of the datapath serve this purpose. Each local memory has a specialized datapath register used as an address register; one of the bus inputs to this address register is replaced by an incrementing feedback path. Like the SILOs found in the Philips VSP [8], this supports the common case of sequential memory accesses. More complex addressing patterns can be generated using registers and ALUs in the datapath.

Input and output data enter and exit RaPiD via I/O streams at each end of the datapath. Each stream contains a FIFO filled with data required or with results produced by the computation. The datapath explicitly reads from an input stream to obtain the next input data value and writes to an output stream to store a result.

External memory operations are carried out independent of the RaPiD array via three I/O streams by placing FIFOs between the array and a memory controller. In addition to carrying out the memory operations, the memory controller generates statically determined sequences of addresses for each stream. If the datapath reads a value from an empty FIFO or writes a value to a full FIFO, the datapath is stalled until the FIFO is ready.

2.2 Control Path

For the most part, the structure of a pipeline is statically configured. However, there are almost always some pipeline control signals that must be dynamic. For example, constants are loaded into datapath registers during initialization but then remain unchanged. The load signals of the datapath registers thus take on different values during initialization and computation. More complex examples include double-buffering the local memories and performing data-dependent calculations.

The control signals are thus divided into static control signals provided by configuration memory as in ordinary FPGAs, and control signals which can be dynamically programmed on every cycle. RaPiD is programmed for a particular application by first mapping the computation onto a datapath pipeline. The static programming bits are used to construct this pipeline and the dynamic programming bits are used to schedule the datapath operations over time. These dynamic control bits are provided by a small pipelined control path, not by the more typical local microprogrammed, SIMD, or VLIW control.

Dynamic control is implemented by inserting a few “context” bits each cycle into a pipelined control path that parallels the datapath. This context contains all the information required by the various pipeline stages to compute their dynamic control signals. The control path contains 1-bit segmented buses similar in structure to the datapath buses, as shown in Figure 2. (Signals which can be dynamic but do not need to change during a particular computation are connected to the static zero line.) Control values are inserted by a global pipeline controller at one end of the control path and are passed from stage to stage where they are applied to the appropriate control signals. Since applications generally use only a few dynamic control signals and use similar pipeline stages, the number of control signals in the control path is relatively small.

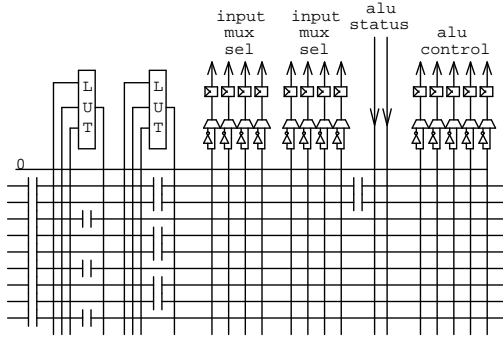


Figure 2: *Dynamic control generation for part of a RaPiD cell; these control buses are one bit wide.*

Each dynamic control signal is derived from the information contained in the control path. Usually the signal is simply connected to one of the bits in the control path, but in more complex cases lookup-tables embedded in the control path are used to compute the control signal based on more information including bits in the control path, status from ALUs in the datapath, or feedback when implementing simple FSM controllers. The generation of dynamic control is illustrated in detail in the applications that follow.

2.3 RaPiD-1 Design Features

Most of the design and layout of the RaPiD-1 chip, the first implementation of the RaPiD architecture, is complete. This section presents those details of RaPiD-1 useful in understanding the performance results discussed for each application presented in the following sections.

RaPiD-1’s datapath is based on 16-bit fixed-point integers; to accommodate this, the multipliers can be statically programmed to shift their 32-bit output arbitrarily. Each RaPiD-1 cell contains three ALUs, one multipliers, and three 32-word local memories. Fourteen tracks are provided for the segmented data

buses, which are supplemented by the zero and feedback inputs available to each functional input. The 16 cells each have the functional units shown in Figure 1, in addition to control logic and up to 15 control buses. The RaPiD-1 array is designed to be clocked at 100MHz, and reconfiguration time for the array is conservatively estimated to be 2000 cycles.

3 Programming Model

Mapping applications to RaPiD involves designing the underlying datapath and providing the dynamic control required for the different parts of the computation. The control design can be complicated because control signals are generated at different times and travel at different rates. We have designed the RaPiD B programming language to accommodate these control patterns. Our RaPiD B compiler which produces a placed and routed implementation along with the dynamic control program is nearly complete. This section first describes a FIR (Finite Impulse Response) filter, a simple application useful for illustrating some of the basic features of RaPiD. It then briefly presents the timing models used by RaPiD B and by the remainder of this paper.

3.1 FIR Filter Computation

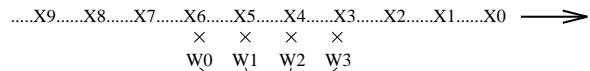
Digital FIR filters are used in many signal processing applications, typically for eliminating unwanted frequency components from a signal. Figure 3a gives a specification for a FIR filter with $NumTaps$ taps and $NumX$ inputs. The filter weights are stored in the W array, the input in the X array, and the output in the Y array (starting at array location $NumTaps - 1$). Figure 3b shows the entire computation required for a single output of a 4-tap FIR filter.

```

for i := NumTaps-1 to NumX-1
  Y[i] := 0
  for j := 0 to NumTaps-1
    Y[i] := Y[i] + X[i-j]*W[j]
  end
end
end

```

(a)



(b)

Figure 3: *FIR filter. (a) Algorithm. (b) Computation for $NumTaps=4$ and $i=6$.*

The circuit in Figure 4a performs the entire computation for one output value in a single cycle; it is easily obtained by unrolling the inner loop of the program

in Figure 3a. Unfortunately, the circuit shown in Figure 4a has poor performance characteristics (note the combinational path through all of the adders, which scales linearly with the number of weights). A retimed version of this circuit is shown in Figure 4b; the retimed circuit performs substantially better than the original, particularly for larger computations.

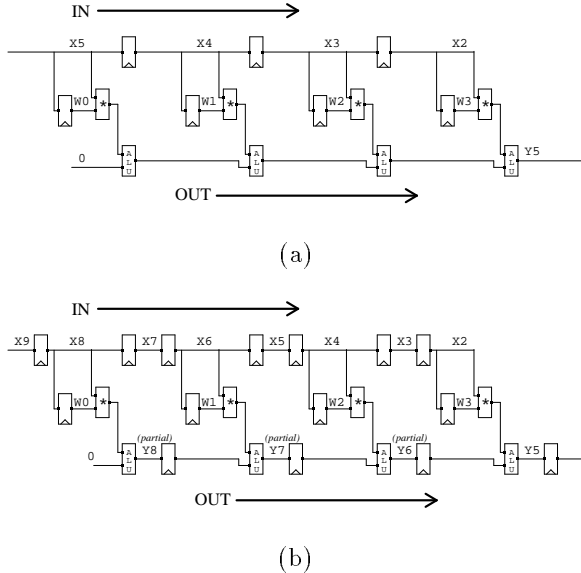


Figure 4: *Schematic diagrams for four-tap FIR filter (a) as viewed in RaPiD B, grouping related computation and (b) as a high-performance pipelined implementation.*

Specifying this retimed circuit directly is difficult because of the complexity of the relative timing of the internal data and control signals. It is much easier to specify the computation somewhat naively as in Figure 4a, knowing that retiming can transform it into a high-performance, pipelined circuit. This becomes particularly evident in circuits with more complicated control, and when more aggressive steps, such as using the pipeline stage available in RaPiD’s multiplier, are needed to achieve the desired performance. Therefore, the RaPiD B compiler retimes the resulting netlist based on [6].

All of the applications presented in the following sections have been specified in a preliminary version of RaPiD B and simulated to validate the implementations described and the accompanying cycle count. For ease of explanation, the computations shown throughout this paper are shown before the full retiming performed by the RaPiD B compiler. A preliminary version of the RaPiD B toolset is nearly complete, including compilation, retiming, control synthesis, and full placement and routing of the resulting RaPiD circuit.

4 FIR Filter Implementation

4.1 Simple Case

As with most applications, there are a variety of ways to map a FIR filter to RaPiD. The choice of mapping is driven by the parameters of both the RaPiD array and the application. For example, if the number of taps is less than the number of RaPiD multipliers, then each multiplier is assigned to multiply a specific weight. The weights are first preloaded into datapath registers whose outputs drive the input of a specific multiplier. Pipeline registers are used to stream the X inputs and Y outputs. Since each Y output must see $NumTaps$ inputs, the X and Y buses must be pipelined at different rates. Figure 5a shows one cell of the FIR filter (several stages are shown in Figure 4b) with the X input bus doubly pipelined and the Y input bus singly pipelined.

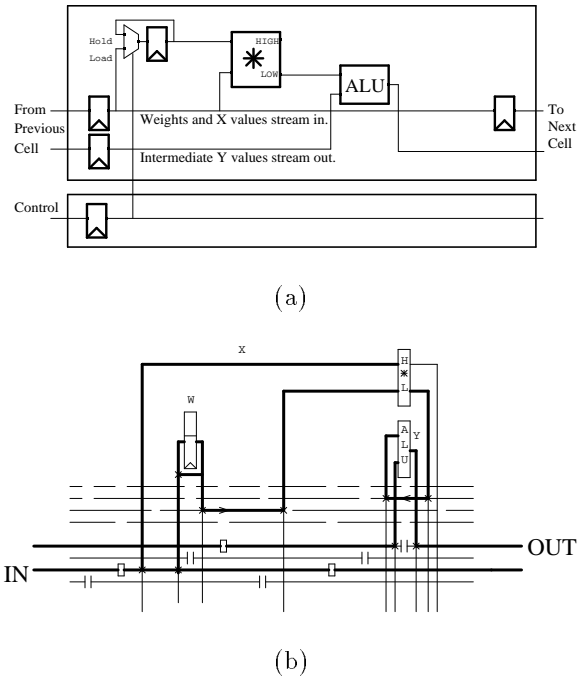


Figure 5: (a) *Netlist for one cell of the simple FIR filter.* (b) *One tap of the FIR filter mapped to the RaPiD array (this is replicated to form more taps).*

This implementation maps easily to the RaPiD array, as shown for one tap in Figure 5b. For clarity, all unused functional units are removed, and used buses are highlighted. The bus connectors from Figure 1 are left open to represent no connection and boxed to represent a register. The control for this mapping consists of two phases of execution: loading the weights and computing the output results. In the first phase, the weights are sent down the IN double pipeline along with a singly pipelined control bit which connects the

input of each datapath register to the IN bus. When the final weight is inserted, the control bit is switched, and the input is connected to the feedback line. Since the control bit travels twice as fast as the weights, each datapath register will hold a unique weight. No special signals are required to begin the computation; the second phase implicitly starts when the control bit goes low.

4.2 Increasing the Number of Taps

If the number of taps exceeds the number of RaPiD multipliers, the multipliers must be time-shared between several taps. This can be achieved in RaPiD by using a local memory to store several weights per stage. Figure 6 shows our implementation for this mapping. Unlike the simple case, we make the arbitrary choice for doubly pipelining the Y output values and singly pipelining the X input values.

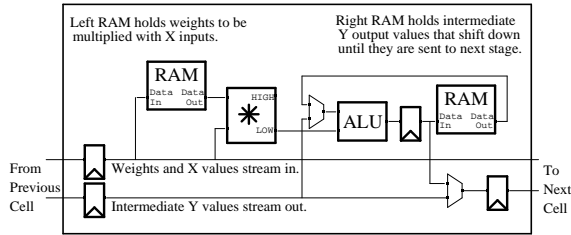


Figure 6: *Netlist for one cell of extended FIR filter. The top pipelined bus streams in the X inputs (the weights during initialization) while the bottom bus streams out the intermediate Y values.*

As a new X is read from external memory, the first stage replicates it and presents it to the input datapath for several cycles. Each stage can multiply this X by its weights in turn and add it to one of its stored intermediate values. At this point a new X value will be fetched from memory and the cycle repeats.

There are the same number of intermediate values as there are weights per stage. These intermediate values are stored in a second local memory. Let's examine the stage holding weights W_{55} , W_{54} , W_{53} , and W_{52} (four taps per stage). A new input value X_{20} appears on the input datapath. In four cycles the partial sums for Y_{75} , Y_{74} , Y_{73} , and Y_{72} will be computed. These are stored in that order in the local memory holding the intermediate values. At this point, X_{20} moves to the next pipeline stage followed by the intermediate value Y_{72} . The next input, X_{21} , appears on the input datapath along with the intermediate value Y_{76} from the previous stage. Now the partial sums for Y_{76} , Y_{75} , Y_{74} , and Y_{73} are computed.

4.3 FIR Performance

When the number of taps is a multiple of 16 the weights can be partitioned evenly across the stages

and the allocated functional units are fully utilized. RaPiD-1 (Section 2.3) can therefore operate at very near its peak performance of 1.6 GOPS (where GOPS is a billion multiply-accumulates per second).

5 Discrete Cosine Transform

The discrete cosine transform (DCT) is used frequently in signal processing and graphics applications. For example, the 2-D DCT is used in JPEG/MPEG data compression to convert an image from the spatial domain to the frequency domain. A 2-D DCT can be decomposed into two sequential 1-D DCTs. We first describe how the 1-D DCT can be computed on RaPiD and then show how two 1-D DCTs can be composed to perform a 2-D DCT.

5.1 1-D DCT

An N -point 1-D DCT partitions an input vector A into N -element sub-vectors, and for each resulting sub-vector A_h computes

$$y_{hi} = \sum_{n=0}^{N-1} a_{hn} \cos \frac{\pi i}{2N} (2n+1) \quad (1)$$

for $0 \leq i \leq N-1$, where a_{hn} is the n -th element of sub-vector A_h (and the $(hN+n)$ -th element of vector A).¹ The N^2 cosine terms are constant over all sub-vectors and hence can be read once as precomputed weights W where $w_{ni} = \cos \frac{\pi i}{2N} (2n+1)$. This reduces Equation 1 to

$$y_{hi} = \sum_{n=0}^{N-1} a_{hn} w_{ni}, \quad (2)$$

for $0 \leq i \leq N-1$. By viewing input vector A and weights W as matrices \mathbf{A} and \mathbf{W} , Equation 2 reduces to the matrix multiply $\mathbf{Y} = \mathbf{A} \times \mathbf{W}$. Thus, to compute a 1-D DCT, RaPiD performs a matrix multiply.

To implement an 8 point 1-D DCT on an 8×8 input matrix \mathbf{A} (i.e. a 64-element vector), the entire 8×8 weight matrix \mathbf{W} is stored in RaPiD's local memories, one column per cell. Each cell of the resulting pipeline is configured as shown in Figure 7. The \mathbf{A} matrix is passed through the array in row-major order. Within each cell, the local memory address is incremented each cycle, and a register accumulates the dot product of the stored column and the incoming row. When a cell receives the last element of a row, the resulting product is passed down an output pipeline, the address is cleared, and the cell is ready to compute the product of the next row on the next cycle. This effectively computes the matrix multiply of $\mathbf{A} \times \mathbf{W}$.

¹To produce the final DCT result each y_{hi} must be multiplied by $\sqrt{\frac{2}{N}} E_i$ where $E_i = \frac{1}{\sqrt{2}}$ if $i = 0$ and $E_i = 1$ otherwise. For our purposes we ignore this scaling factor and focus on the computation of y_{hi} .

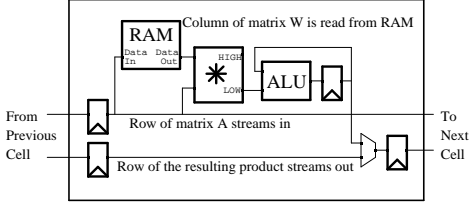


Figure 7: *Netlist for one cell of a matrix multiply. The top pipelined bus streams in the \mathbf{A} matrix (in row-major order) while the bottom bus streams out the resulting matrix product (also in row-major order). The top bus also streams the \mathbf{W} columns into the local memories prior to the computation.*

5.2 2-D DCT

An $N \times N$ 2-D DCT partitions an input matrix into sub-matrices of size $N \times N$, and for each resulting sub-matrix \mathbf{A} , computes

$$y_{ji} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} a_{mn} \cos \frac{\pi i}{2N} (2m+1) \cos \frac{\pi j}{2N} (2n+1) \quad (3)$$

for $0 \leq i, j \leq N-1$.² As with the 1-D DCT, Equation 3 is reduced using the N^2 precomputed \mathbf{W} weights, yielding

$$y_{ji} = \sum_{m=0}^{N-1} \sum_{n=0}^{N-1} a_{mn} w_{mi} w_{nj} \quad (4)$$

for $0 \leq i, j \leq N-1$. Extracting w_{mi} from the inner summation leaves

$$z_{mj} = \sum_{n=0}^{N-1} a_{mn} w_{nj}, \quad (5)$$

and thus

$$y_{ji} = \sum_{m=0}^{N-1} z_{mj} w_{mi} \quad (6)$$

for $0 \leq i, j \leq N-1$.

As seen in Equation 5 and Equation 6, both z_{mj} and y_{ji} are equivalent to $N \times N$ matrix multiplies. However, since the z_{mj} values are produced in row-major order but required in column-major order, the results from the z_{mj} DCT must be transposed prior to computing y_{ji} as illustrated in Figure 8. In addition, since both input streams are read in row-major order, it might be desirable to produce row-major output (potentially reducing memory stalls), requiring yet another transform (i.e. output y_{ij} instead of y_{ji}). The resulting computation is $((\mathbf{A} \times \mathbf{W})^T \times \mathbf{W})^T$.

²To produce the final DCT result each y_{ji} must be multiplied by $\frac{2}{\sqrt{N}} E_i E_j$. As with 1-D DCT, we ignore this scaling factor and focus on the computation of y_{ji} .

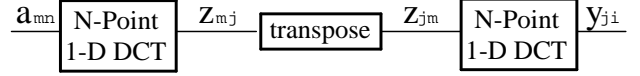


Figure 8: *2-D $N \times N$ DCT*

We show the implementation of an 8×8 2-D DCT on a 16-cell RaPiD array. Consider an $M \times N$ image and an 8×8 weight matrix \mathbf{W} . First, the image is divided into $\frac{MN}{64}$ sub-images of size 8×8 . The computation for each sub-image \mathbf{A} is outlined in Figure 9.

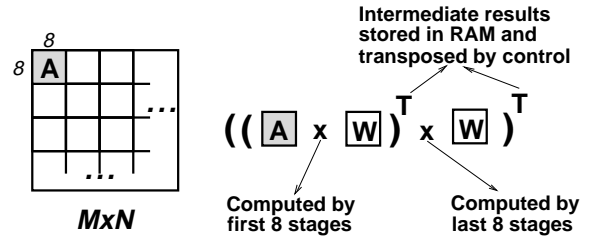


Figure 9: *To compute 2-D DCT, an $M \times N$ image is partitioned into 8×8 sub-images. RaPiD computes each 1-D DCT by multiplying the sub-image by an 8×8 weight matrix.*

Since a 2-D DCT performs two multiplies by the same weight matrix, \mathbf{W} is loaded only once: one column per cell in both the first 8 cells and last 8 cells. The transpose in between matrix multiplies is performed with two local memories per cell: one to store products of the current sub-image and the other to store the products of the *previous* sub-image. During the computation of the current sub-image, the transpose of the previous sub-image computation is passed to the next 8 cells. The datapath for one RaPiD cell of a 2-D DCT is shown in Figure 10.

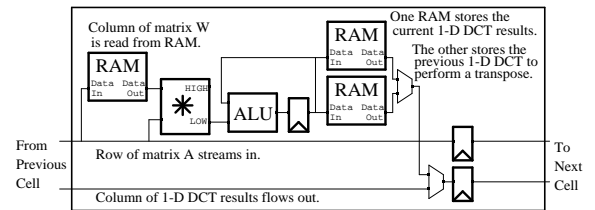


Figure 10: *Netlist for one cell of 2-D DCT. The top pipelined bus streams in the \mathbf{A} matrix while the bottom bus streams out resulting 1-D DCT, transposed. The top bus also streams the \mathbf{W} columns into the local memories prior to the computation.*

5.3 DCT Control

Prior to computation, a 2-D DCT must load the \mathbf{W} matrix into the local memories, one column per stage. To take advantage of pipelined control, weights are passed down a data-bus in *row-major* order, while a control signal, traveling twice as slow as the data, raises the write signal of the appropriate local memories. As a result, all weights of the DCT can be preloaded using a single control bus. Most RaPiD control signals fit into such a simple, pipelined model since an operation occurring in one RaPiD stage usually occurs in the next stage on the next cycle.

Sometimes control is required which does not fit into the simple, pipelined model. At the end of the first 1-D DCT computation, results are stored one column per stage. To flow these results out in *column-major* order (that is, perform the transpose), the first local memory must be completely emptied onto the output bus, followed by the second, third, etc. Hence, the “empty” control signal must stay on for eight consecutive cycles in the first stage, and then eight cycles in the second stage, etc. Possible solutions include dedicating a control bus to every stage or using one control bus with eight registers per stage. The solution requiring the fewest resources configures two buses and one 3-LUT per stage as a simple finite state machine, as shown in Figure 11.

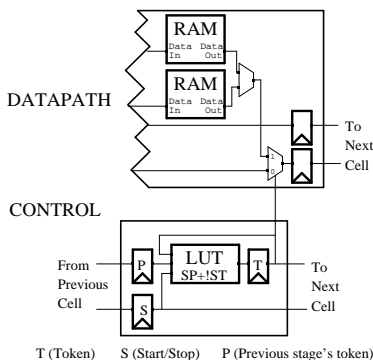


Figure 11: A simple state machine performs the transpose using two buses, one LUT, and three registers per stage.

Three control registers are used in the state machine: T is the token, S is the start/stop bit, and P is the previous stage’s token delayed by a cycle. The LUT is configured as a multiplexer of P and T with select bit S (i.e. $T = S \& P + !S \& T$). If S is low, the token is held; if S is high, the token is passed to the next stage. When a stage has a token, its results are emptied from a local memory onto the output bus. This operation repeats in each consecutive stage, effectively transposing the 1-D DCT results.

To initiate the transpose, the stream controller places a one into the first P register every 64 cycles

and a one into the first S register every 8 cycles. Notice that the token hold length is solely determined by the frequency of the start/stop signal and does not affect the number of control buses, LUTs, or registers needed. Thus, the size of this state-machine control is fixed no matter how long each stage must hold a token.

5.4 DCT Performance

A 2-D DCT performs many consecutive 8×8 matrix multiplies, allowing initialization, finalization, and re-configuration times to be small compared to the total computation performed. For example, RaPiD-1 (Section 2.3) incurs a setup overhead of only 0.5% to compute the 2-D DCT of a 720×576 image. As a result, RaPiD-1 performs very close to its peak of 1.6 GOPS on 2-D DCT (where GOPS is a billion multiply accumulates per second).

6 Motion Estimation

Motion estimation is used in video data compression to reduce the amount of data required to represent a video frame. In most cases, objects do not move very much from one frame to the next. In motion estimation, a block in a frame is represented by the address of the most similar nearby block in the previous frame plus the differences between the two blocks. This section describes implementing motion estimation on RaPiD.

Motion estimation has few data dependencies, providing flexibility in the order of computations and greater parallelism. RaPiD favors computations that are not memory bound. The prodigious amount of computation and few memory accesses make motion estimation an ideal candidate for RaPiD.

To compute the motion estimation of an $M \times N$ reference image, the image is divided into $\frac{MN}{64} 8 \times 8$ reference blocks (RB). The reference blocks are compared with blocks of a prior video frame, the query image. For each reference block RaPiD computes the minimum absolute *block difference* (point-to-point difference) of all possible positions of the RB within a 24×24 query window (QW) of the query image, as shown in Figure 12. The result is a vector which points to the RB yielding the minimum block difference.

6.1 Motion Estimation Implementation

With a 16-stage RaPiD array we implement motion estimation using 16×16 super reference blocks, which are comprised of four 8×8 reference blocks, and 32×32 super query windows. The super RB and a 32×16 section of the query window are stored in RaPiD’s local memories, one column per stage. This mapping yields

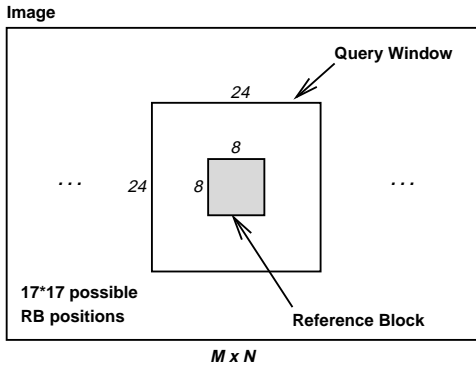


Figure 12: The image is partitioned into 8×8 RBs. Motion estimation of the RB within a 24×24 QW is determined by finding the minimum block difference for all positions of the RB within the QW.

the best reuse of RB and QW values for the available local memory. A stage of the resulting pipeline is shown in Figure 13.

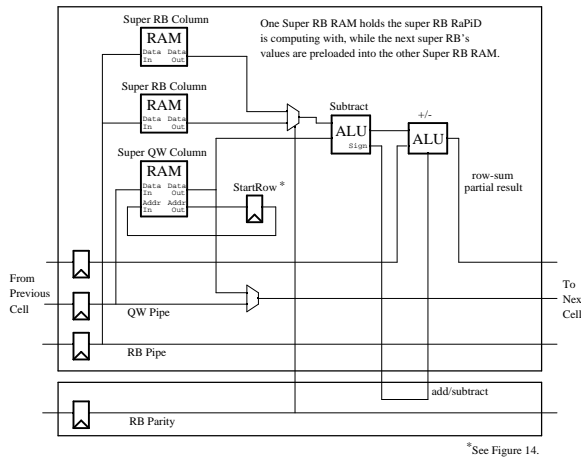


Figure 13: Cell configuration for motion estimation compute stages. 16-bit data and 1-bit control lines are drawn in separate boxes. To achieve an absolute-difference the sign bit of the Subtract ALU controls the function of the +/- ALU.

The block difference between a super RB and super QW is computed row by row. For each row, a stage performs an absolute-difference and accumulates the result with the absolute-difference of the prior stage. This operation happens in the same way as the FIR filter of Section 4. The last stage totals all of the row sums to produce the block difference and determines the minimum block difference for each RB of the super RB.

The netlist for motion estimation, presented in Figure 13, shows how two dynamic control lines control an ALU and super RB local memory selection.

The absolute-difference-accumulate operation is implemented by controlling the function of the +/- ALU with the sign of the subtract ALU.

The local memory used for the super RB is double-buffered, with one local memory used for the current computation while the other is being preloaded with the next super RB. The parity control signal is used to determine which local memory to use for computation and which to use for preloading. The parity signal toggles when a motion vector for the current super RB is output.

6.2 Motion Estimation Data Flow

To obtain the most reuse of data we perform block differences in column-major order. That is, the super RB starts in the upper right corner of the super QW and proceeds down the rows before shifting left one column.

A left shift of the super RB is implemented by shifting the super QW columns right, to the next stage. When a super QW value is no longer needed, the value is shifted to the next stage and a new value is shifted in from the prior stage. The first stage gets new super QW values from the QW input stream.

Super QW values are reused between block differences by storing the address of the starting row of the super QW in the StartRow register (Figure 14). When a block difference completes, the QW column local memory address is set to StartRow and StartRow is incremented. StartRow is reset when the super RB is shifted left one column.

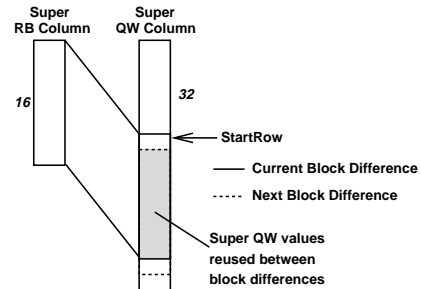


Figure 14: The super RB column shifts down through super QW column, performing a block difference at each step. StartRow is the address of the first row of the block difference.

Moving the super RB from right to left allows super QW values to be reused between sets of block differences. Figure 15 shows how the last columns of the current super QW are the first columns used in the super QW of the next super RB computation. This data motion removes the need to preload super QW values for the next set of block differences.

The only time data loading stalls computation is the beginning of a row of super RBs. In this case

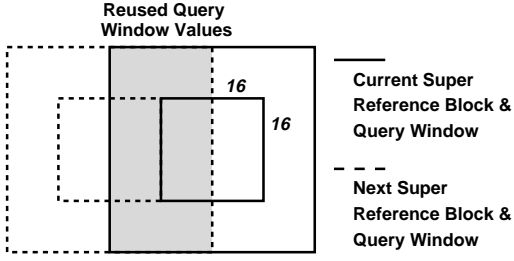


Figure 15: The last super QW columns used to compute motion estimation for a super RB are reused in the computation for the next RB.

the required super QW values were not used with the prior super RB and must be loaded. The next section shows that the cost is minor, being amortized over a long computation.

6.3 Motion Estimation Performance

Motion estimation is not a memory bound computation and with our implementation no memory stalls are encountered. The cycles not spent computing absolute-difference-accumulation operations are due to initialization, finalization, reconfiguration, and the loading of super QWs. For an image of size 720×576 , using RaPiD-1, loading the super QW costs 18,432 cycles for motion estimation of one frame³. The overhead of loading and reconfiguration time take less than 0.03% of the total number of cycles. As a result, a RaPiD-1 array performs close to its peak speed of 1.6 GOPS (where GOPS is a billion absolute-difference-accumulates per second).

The speedup of motion estimation scales well as the data size grows and with future versions of RaPiD. As data size grows, the cycles used to load super QWs will grow linearly, while the cycles spent in computation grow with the square of the data size. Thus as the data size grows a larger percentage of cycles will be spent computing.

Future versions of RaPiD will have more stages and larger local memories per stage, increasing the number of RBs per super RB and thus the amount of parallelism. Typical images also use 8-bit data, allowing us to double gauge RaPiD’s 16-bit data path, gaining another factor of two in speedup.

7 Parametric Curve Generation

This section describes how arbitrary 2-D Bézier curves with four control points⁴ can be computed by RaPiD using Apex, an architecture for generating a large class

³The super QW must be preloaded 576/16 times and a preload takes $16 * 32$ cycles, resulting in 18,432 cycles.

⁴With very little additional effort this can be changed to Bézier curves of arbitrary dimension and with up to six control points on a 16-cell RaPiD array.

of parametric curves and surfaces [2]. Apex differs from the previous applications in that it maps a triangular data-flow onto RaPiD as shown in Figure 16. Each node in the tree performs a weighted average on the two inputs values and passes the result to the parent node. In symbolic form this is equivalent to

$$V_s(t) \leftarrow (1-t)V_{\text{left}} + tV_{\text{right}} = V_{\text{left}} + (V_{\text{right}} - V_{\text{left}})t$$

The root node produces a new point of the Bézier curve for each t . The nodes are mapped onto the RaPiD stages in the order indicated by the numbers inside the nodes (Figure 16). This particular mapping minimizes the communication between nodes.

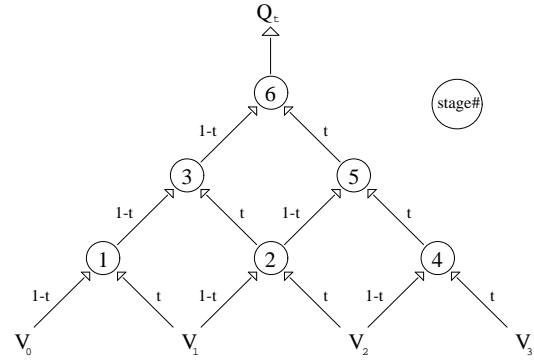


Figure 16: Data-flow graph for computation of the Bézier curve Q_t described by the control points V_i . Each node performs a weighted average (weights are edge labels) of its two inputs.

7.1 Apex Implementation

The algorithm can be split into initialization and computation. During initialization, the control points are loaded (e.g. stages 1, 2, and 4 in Figure 16) and a Δt increment is specified for t . Then the repetitive computation starts in which each node increments its private copy of t by Δt and performs the required computation. During the computational phase, no further external inputs are required.

Computing a 2-D Bézier curve produces two coordinate values per point. The two values can be computed independently. Since we only need six stages per coordinate value (see Figure 17), both can be computed in parallel using a total of twelve stages.

The accuracy and resolution is limited by that t and Δt which in our current implementation is represented by a 16 bit register. The value t can be computed in the first stage using two registers (i.e. 32 bits) to substantially reduce the forward differencing errors. It would then propagate down the pipeline.

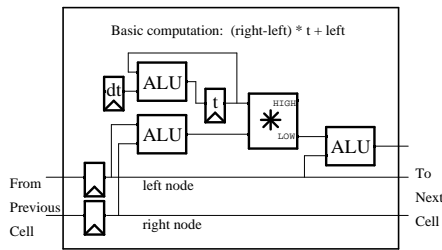


Figure 17: Netlist for one cell of Apex. The *dt* register is loaded from a datapath (not shown) before computation begins. Leaf nodes have two additional registers holding the constant control points.

7.2 Apex Performance

Apex outputs a new point of the Bézier curve every cycle with relatively small initialization overhead. If we assume that 100 1000-point curves are displayed before reconfiguration is necessary, the setup overhead is only 0.2% for RaPiD-1 (Section 2.3) and it would perform at nearly 1.2 GOPS (where one OP is one weighted average). This is close to peak performance with the small loss in performance due to the fact that four cells are not used in the computation.

8 Conclusion and Future Directions

RaPiD represents an efficient configurable computing solution for regular computationally-intensive applications. In this paper, we have described how four different applications are mapped to the RaPiD array. These applications require a particular set of architectural features provided by RaPiD. We believe this feature set enables RaPiD to perform a wide range of different computations. By combining the appropriate amount of static and dynamic control, RaPiD achieves substantially reduced control overhead relative to FPGA-based and general-purpose processors. RaPiD is optimized for highly predictable and regular computations, reducing the control overhead. The assumption is that RaPiD will be integrated closely with a RISC engine on the same chip. The RISC would control the overall computational flow, performing the unstructured computations which it does best, while farming out the heavy-duty, brute-force computation to RaPiD.

Several challenges remain. The range of RaPiD applications needs to be extended, and integrated applications comprising different computations need to be investigated. The RaPiD B programming model needs to be evaluated and new compiler optimizations implemented. Finally, we would like to investigate how parallel language and compiling methods can be applied to programming RaPiD applications at a higher level.

Acknowledgments

We would like to thank Larry McMurchie and Chris Fisher for their contributions to the RaPiD project.

References

- [1] J. M. Arnold et al. The Splash 2 processor and applications. In *Proceedings IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 482–5. IEEE Comput. Soc. Press, 1993.
- [2] T. DeRose et al. Apex: two architectures for generating parametric curves and surfaces. *Visual Computer*, 5:264–76, 1989.
- [3] C. Ebeling, D. C. Cronquist, and P. Franklin. RaPiD—reconfigurable pipelined datapath. In R. Hartenstein and M. Glesner, editors, *6th International Workshop on Field-Programmable Logic and Compilers*, Lecture Notes in Computer Science, pages 126–135. Springer-Verlag, September 1996.
- [4] H. Kung. Let’s design algorithms for VLSI systems. Technical Report CMU-CS-79-151, Carnegie-Mellon University, January 1979.
- [5] P. Lee and Z. M. Kedem. Synthesizing linear array algorithms from nested FOR loop algorithms. *IEEE Transactions on Computers*, 37(12):1578–98, 1988.
- [6] C. E. Leieron and J. B. Saxe. Retiming synchronous circuitry. *Algorithmica*, 6:5–35, 1991.
- [7] D. I. Moldovan and J. A. B. Fortes. Partitioning and mapping algorithms into fixed size systolic arrays. *IEEE Transactions on Computers*, C-35(1):1–12, 1986.
- [8] K. A. Vissers et al. Architecture and programming of two generations video signal processors. *Microprocessing & Microprogramming*, 41(5-6):373–90, 1995.
- [9] J. E. Vuillemin et al. Programmable active memories: reconfigurable systems come of age. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 4(1):56–69, 1996.