Chapter #1

Propagating Constants Past Software to Hardware Peripherals in Fixed-Application Embedded Systems

Greg Stitt, Frank Vahid
Department of Computer Science and Engineering, University of California, Riverside

Abstract: Many embedded systems include a microprocessor that executes a single program for the lifetime of the system. These programs often contain constants used to initialize control registers in peripheral hardware components. Now that peripherals are often purchased in intellectual property (core) form and synthesized along with the microprocessor onto a single chip, new optimization opportunities exist. We introduce one such optimization, which involves propagating the initialization constants past the microprocessor to the peripheral, such that synthesis can further propagate the constants inside the peripheral core. While constant propagation in synthesis tools is commonly done, this work illustrates the benefits of recognizing initialization constants from the software as really being constants for hardware. We describe results that demonstrate 2-3 times reductions in peripheral size, and 10-30% savings in power, on several common peripheral examples.

Key words: Cores, system-on-a-chip, embedded systems, synthesis, low power, constant propagation, platforms, tuning, intellectual property.

1. INTRODUCTION

Embedded system designers are increasingly composing their designs from pre-designed intellectual-property cores, integrating those cores into a single chip model as shown in *Figure 1*, and then fabricating a chip [3]. A core is a description of a system-level component, like a microprocessor, memory, or peripheral component like a direct-memory access (DMA) controller or universal asynchronous receiver/transmitters (UART). Cores come in three forms. A soft core is a synthesizable hardware description language (HDL) model. A firm core is a structural HDL model. A hard core is a technology-specific layout. Many commercial core libraries now exist, e.g., [4], and core standards are evolving rapidly [9].

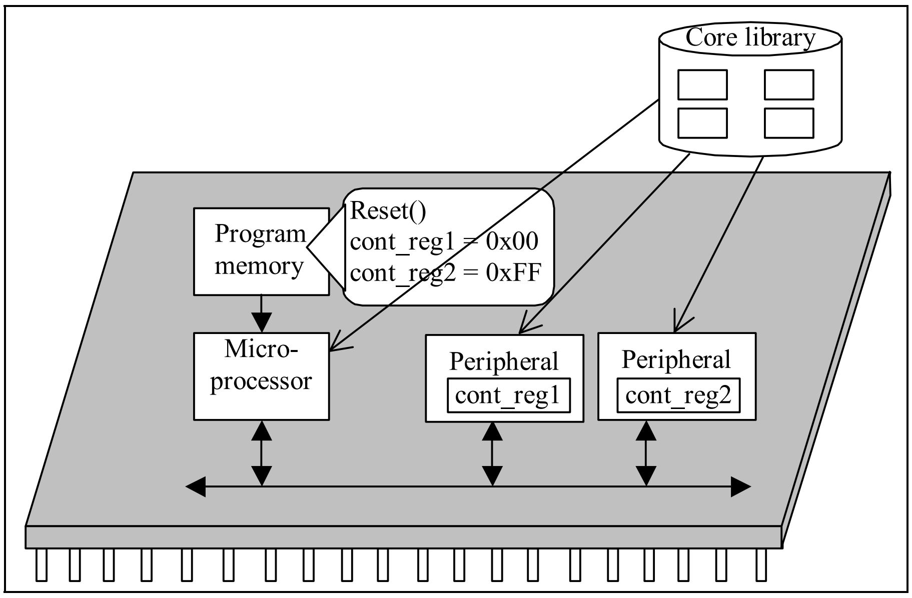


Figure 1. Core-based embedded system design.

A designer gains many advantages from building a system from standard cores, such as a standard DMA controller or UART. Most importantly, the designer gains improved time-to-market due to familiarity with the standard core and compatibility with development tools. Such standard cores typically come with parameters [8]. Some are pre-fabrication parameters, which are set by a designer before synthesis, thus influencing the synthesis results. Such parameters are typically achieved using generics or constants in a hardware description language (HDL), but can also be achieved using module generators, which generate unique HDL models depending on the parameter selection. For example, a JPEG decompression core might by synthesizable to have either 12 or 16-bit resolution. Synthesizing for 12-bit resolution would yield a smaller core.

Other parameters, in contrast, are post-fabrication parameters, set only after the core has been synthesized. Such parameters' settings are typically stored in registers or non-volatile memory inside the core. They are more commonly referred to as software configurable parameters. For example, a DMA controller will have a base register to indicate the starting address in memory from which the controller should move data, and a block size register to indicate the number of words that should be moved. An arbiter core might have a register whose setting determines whether arbitration uses a fixed or rotating priority scheme.

We make the observation that an embedded system typically runs a single program that never changes – the application is fixed. In fact, in many cases that program cannot be changed, because it may be burned into ROM (using mask-programmed ROM) that appears with the microprocessor and peripherals on a single chip to reduce chip cost, size and power (at the expense of less flexibility).

A typical embedded system will execute a boot program upon system reset, and this program will, among other things, set these software configurable parameters in the system's peripherals, as shown in *Figure 1*. However, if the embedded system's program never changes, then those register values never change during the execution of the embedded system. For example, a particular embedded system may use a DMA controller to repeatedly send data directly from an array, of size 48 and starting from memory location 100, to a display device. The system's boot program may set the DMA controller base register to 100, and the block size register to 48. These values will never change for the life of the embedded system.

Previously, when systems were built using discrete off-the-shelf integrated circuits, such software configuration was necessary. However, since today's systems are being built with cores, we now have an optimization opportunity that did not previously exist. Specifically, for an embedded system whose program does not change, the values to which the software configurable peripheral parameters are being set are really constants. As compiler writers are well aware, constants provide excellent optimization capability, through the well-known compiler optimization known as constant propagation and constant folding[1][10]. Such propagation consists of replacing a variable holding a constant by the constant itself. This replacement can result, for example, in branch conditions that always evaluate to false, resulting in turn in dead code that can then be eliminated. It can also enable compile-time evaluation of expressions.

Such dead code resulting from constant propagation is especially common when propagating constants into subroutines through the subroutine's parameters. While the subroutine may have been designed to handle a variety of sets of parameters, a particular program may only call the subroutine with certain constant values for those parameters, resulting in much dead code in the subroutine.

We can think of a peripheral core as similar to a subroutine, in fact, as a subroutine that has been implemented using additional hardware. The core may have been designed to handle a variety of sets of software configurable parameters. However, a particular program may only use the core with certain constant values for those parameters, resulting in much dead code in the core. We therefore propose a deeper propagation of constants than performed by compilers. In particular, we propose to propagate those constants beyond the microprocessor's program, to the microprocessor's peripheral cores – essentially propagating those constants all the way to peripheral hardware. Those constants would then be fed into the synthesis tool being used to synthesize the cores. The synthesis tool could then perform constant propagations and dead code elimination during synthesis, where the code here refers to the core's HDL description. Most commercial synthesis tools already include such compiler optimizations, but those optimizations are only applied to the pre-fabrication parameter constants. We will show that much benefit would come from enabling the synthesis tool to recognize the post-fabrication parameter values as constants also.

The end result of such propagation is that the synthesized core will be optimized for the particular program that is using the core, something we refer to as architecture tuning [8]. By optimized, we mean that the core will have fewer gates, and consume less power, than a standard version of the same core. Reducing size is important since such reduction can increase chip yield and reduce chip cost, and many embedded systems are extremely cost sensitive, especially those being manufactured in high volumes. Reducing power is important since many embedded systems operate on batteries or draw power from very limited sources, and so power reduction is an important design criterion.

In the following sections, we introduce the concept of propagating constants past software to hardware peripheral cores. After an introductory example, we'll describe common core parameters that are candidates for constant propagation, discuss methods for achieving such propagation, and highlight experiments showing the size and power reductions possible. The results motivate future work on developing tools that introduce some cooperation between the compilers and the synthesis tools being used in developing a system-on-a-chip from cores.

2. EXAMPLE

As a simple illustration of propagating constants to hardware, let us consider a trivially simple peripheral core that has two parallel ports. Each port can be configured to be an input port or an output port. A VHDL description of part of the core is shown in *Figure 2*(a). The core description declares a control register *cont_reg* with two bits. The first bit makes port A an output port when set to 0, and an input port when set to 1. Likewise, the second bit makes port B an input or output port. The VHDL description begins with initialization of the control register during a reset. Next, it would describe the synchronous monitoring of the bus for an address corresponding to the control register, and the writing of the control register in this case – this code is omitted from the figure. Next, the VHDL description describes the control logic for the tri-state buffers that implement the port direction functionality. Finally, other behavior of the core would be described.

Synthesis converts this soft core to hardware structure, shown in *Figure 2*(b). Note that logic is generated to handle the bus monitoring and the control of the four required buffers.

Now, consider the situation where this core is used in an embedded system and controlled by a microprocessor executing a fixed C program. We might see the following assembly code embedded in the reset routine of the C program:

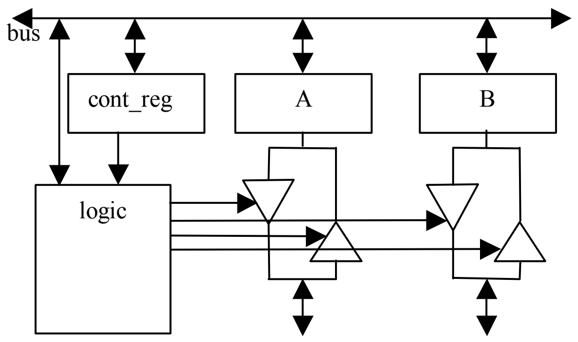
```
OUT cont_reg, #"00000010"
```

Assuming *cont_reg* is the address of the control register in the microprocessor's I/O address space, then this code would write the constant "00000010" onto the peripheral bus, resulting in a 0 being written into *cont_reg*(0) and a 1 into *cont_reg*(1). The peripheral core would thus be configured with port *A* as an output port, and port *B* as an input port. The rest of the C program would then access these ports appropriately.

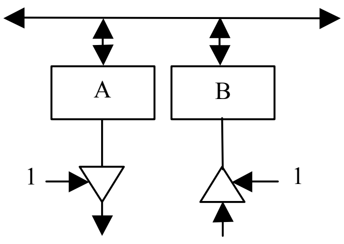
Now, suppose we could somehow propagate the constant "10" into the VHDL description of the core, before the core were synthesized, letting the synthesis tool know that *cont_reg* would be written by that constant and only that constant. If we did this in a way that our synthesis tool could make use of that information, then the synthesis tool would find much dead code in the VHDL description. First, the control register would not be needed, since a constant can be derived directly from power and ground in hardware. Second, the logic to monitor the bus for the control register address and then write the register would not be needed. Third, each buffer control signal if statement would have one branch that was always true and the other always false. Finally, the reset code of the core would not be needed. After all of this dead code is eliminated, the synthesis tool would output the structure shown in *Figure 2*(c). The resulting structure in this case requires less hardware, and would also consume less power due in part to elimination of the bus monitoring.

```
signal cont_reg:
       UNSIGNED(1 downto 0);
-- declarations for A, B and
   buffers omitted.
process(clk, reset)
begin
if (reset) then
   cont_reg = "00";
   A_out <= '1'; A_in <= '0';
   B_out <= '1'; B_in <= '0';
end if;
if rising_edge(clk) then
   -- Code to detect write request
   -- from bus to cont_reg, and
   -- to update cont_reg, omitted
   if (cont_reg(0) = '0') then
       A_out <= '1';
       A_in  <= '0';
   else
        A_in  <= '1';
        A_out <= '0';
   end if;
   if (cont_reg(1) ='0') then
       B_out <= '1';
       B_in  <= '0';
   else
       B_in  <= '1';
       B_out <= '0';
   end if;
end if;
-- Other behavior omitted
end process;
```

(a)



(b)



(c)

Figure 2. A simple example of propagating constants to hardware (a) soft core, (b) synthesized core structure, (c) synthesized core structure after propagating constants cont_reg(0)=0 and cont_reg(1)=1.

3. PARAMETERS IN CORES

We examined a number of common peripheral cores, and found many software configurable parameters that could be candidates for constant propagation. Some common peripheral cores include the Intel 8255A (programmable peripheral interface), the 8237A (DMA controller), and the M16550A (UART – Universal Asynchronous Receiver-Transmitter).

Figure 3 is the block diagram of the 8255A. The 8255A interfaces with a microprocessor on one side, and provides three configurable ports on the other side. Its software configurable parameters include mode of operation, number of ports in use, and direction of each port (input or output). These parameters are set by a microprocessor by writing an 8-bit control word into a control register in the 8255A.

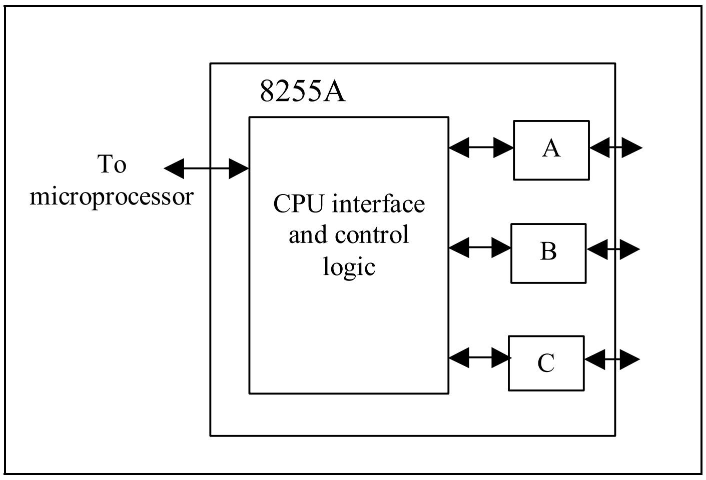


Figure 3. The Intel 8255A parallel peripheral interface.

The 8237A includes even more software configurable parameters, including the number of channels, the type of priority scheme (fixed or rotating) being used to arbitrate between channels, whether each channel operates in single transfer mode or block transfer mode, the starting address and block size for each channel, etc. There are thus several control registers in the 8237A.

Likewise, the UART's parameters include the baud rate, parity type, mode of communication, etc.

In general, peripheral cores tend to have several types of configurable parameters, related to features such as:

- Size of internal data or external data bus
- Number of channels or ports
- Modes of operation
- I/O direction
- Rate of data transfer
- Resolution

Supporting numerous parameters is necessary in order for a peripheral to be applicable in a variety of systems and thus to sell in large quantities. While some parameters appear in a core as user-settable constants or generics, others appear as software configurable control registers. Such software configurability is used in peripherals for several reasons. One reason is that, before the advent of cores, software configuration was the only way to configure a peripheral integrated circuit (IC). A core may thus be modeling a widely-used standard peripheral that was defined in the time of ICs, such as UART and DMA controller cores. A second reason is that, even for cores representing new peripherals, the core designer does not know if the peripheral will be controlled by a microprocessor whose application will not change. If the core were used in a system whose application did change, then constant or generic-based parameters would not be appropriate. Thus, support of software configurable parameters is very common, but results in extra hardware size as well as power consumption.

4. PROPAGATING CONSTANTS FROM SOFTWARE TO HARDWARE

We now describe a method for manually propagating constants across the software/hardware boundary in a core-based synthesis methodology, and discuss potential approaches for automating this method. The method is summarized in *Figure 4*. For each core, the first step is to determine all of the registers in the core that serve as control registers for the various parameters listed in the previous section. Next, for each such control register, we must look for all references to that register in the driving microprocessor program. If the only access to that register is a write with a constant, and this write occurs during the reset or boot routines, as is often the case in embedded systems, then we have a candidate for constant propagation to peripheral cores. We replace the register's declaration in the core by a constant declaration. We delete any behavior that involves detecting and carrying out a write to that register from the peripheral bus. We can then run the synthesis tool on this modified core. A synthesis tool will then detect and eliminate the dead code created by the constants we introduced in the model, and thus result in a simpler synthesized structure. Most modern synthesis tools already carry out standard compiler optimizations like constant propagation, constant folding, and dead code elimination.

```
for each peripheral core P
   for each control register C in P
      for each write, W, to C in processor’s program
         if W consists of a single write, of a constant X,
               in a reset or boot routine, then
            replace C in P by a constant declaration set to X
            delete behavior related to writing C in P
         end if
      end for
   end for
end for
run synthesis as usual
```

Figure 4. Method for propagating constants to peripheral cores.

We can also eliminate the behavior in the microprocessor’s program relating to writing the control register, but this is not always necessary. If we do choose to leave it, then we must ensure that the lack of a response from the core is acceptable. If a response is needed, like an acknowledgement, then we leave such behavior in the core.

The above method has the advantage of being immediately applicable in any existing core-based design process, without any modification to existing tools. Of course, the constant propagation across the software/hardware boundary must be performed manually in the above case. Thus, we describe a potential approach to automating the method. A big help to such automation is if a core design framework is being used. Such frameworks, many of which are commercially available, manage the retrieval and instantiation of cores (e.g., [2][5]). They typically already have support for instantiating cores with specific values for constants or generics (a generic is essentially a parameter whose value must be chosen before instantiation) and for keeping track of all register address assignments in a system of cores.

Thus, modifying such frameworks to handle software configurable parameters can be seen as an extension of an existing method.

One approach to automation would be to extend the software compiler to output a list of external I/O addresses that are assigned a single constant by the program in a reset or boot routine, along with each address' associated constant. This requires that the compiler be aware of the location of those reset or boot routines. Next, each core must have its control registers known to the core framework – this can be done by the framework developer, or the framework user, without too much effort. Furthermore, the framework must know where in the core to find the code that writes the register. Given this setup, the framework can read the contents of the file output by the software compiler, and for each address the framework can then replace the corresponding register declaration by a constant declaration, and delete write behavior from the core, before instantiating the core into the design. Then, synthesis can be run on the instantiated core, and the constants will result in dead code that can be eliminated.

A second approach is possible, and in fact even simpler than the above. In particular, we observe that modern core-based frameworks actually generate the reset or boot code themselves, including the code for initializing peripherals [5]. In other words, suppose a user wishes to instantiate a DMA controller into a system already having a microprocessor and memory. The framework will query the user to ask for the values of software configurable parameters, like transfer mode, base address and block size. The framework then generates the necessary driver software on the microprocessor. The second approach extends the above by having the framework also ask if the software configurable parameter values will ever change, or if instead they are in fact constants. If they are constants, then the framework can withhold generation of the related driver software, and instead directly proceed to instantiate the core with the corresponding register declaration replaced by a constant, and with the register-write behavior deleted.

5. EXPERIMENTS

We performed several experiments to evaluate the size and power savings possible by using our method of propagating constants to peripheral cores. We modelled three popular peripherals as register-transfer level VHDL soft cores: the 8255A programmable peripheral interface, the 8237A DMA controller, and the 16550A UART. Each core model is nearly a fully-functional model. The three soft-core models required 1045, 920 and 1063 lines of VHDL code, respectively. We also obtained a discrete cosine transform (DCT) core (Free-DCT-L) from http://www.opencores.org, which consisted of 910 lines of code. We manually modified these models to eliminate dead code that would have resulted from constant propagation of the software-configurable parameters described below. We synthesized the cores twice, once before and once after dead code elimination, using the Synopsys Design Compiler. Area and power were measured using Synopsys analysis tools, with power measured while running a suite of test vectors for each core. Because we wanted to see first-hand the impact of the constant propagation on the size of the VHDL code, we performed the propagation of the constants and the dead code elimination manually, so we could measure the resulting lines of code.

5.1 8255A Programmable Peripheral Interface

The 8255A had only one configuration register used for selecting the modes of various ports. We examined the impact of propagating constants for three different configurations of this register. Mode0 corresponded to a configuration where port A of the device was used as an output port. Mode1 corresponded to port A being used as an output port with handshaking I/O. Mode2 corresponded to port A being used as a bi-directional port with interrupt I/O. Each situation resulted in a reduction of the number of lines in the model from 1045 to an average of only 415 lines.

Optimizations from constant propagation are shown in the following code:

```
if( cont_reg(4)='1' ) then
    pao <= "ZZZZZZZZ";
    paen <= '1';
elsif( cont_reg(4)='0' ) then
    paen <= '0';
end if;
```

In this example, *cont_reg* represents the control register for the device. By setting the fourth bit to 1, port A is configured as an input port. Note that this is accomplished with tri-state buffers that set the output signal, *pao*, to a disconnected state. *Paen* is used to enable port A for input. If we know that a program will always require port A to be used for output, we can simply replace the example with an assignment of zero to the *paen* signal. This will reduce area by eliminating several tri-state buffers and the logic required to implement the if statement.

Another example is shown below:

```
if( cont_reg(7) = '1' and
    cont_reg(6 downto 5) = "00" ) then
    a_mode <= A_0;
elsif( cont_reg(7) = '1' and
       cont_reg(6 downto 5)= "11" ) then
    a_mode <= A_1;
elsif( cont_reg(7) = '1' and cont_reg(6)= '1' ) then
     a_mode <= A_2;
end if;
```

In this example, the control register is being checked to determine the appropriate mode for port A. Therefore, if we set the control register to a constant value, then *a_mode* is also set to a constant value. We can then propagate the *a_mode* constant into the following code:

```
case( a_mode ) is
    when A_0 =>  -- implements mode 0 for Port A
     …
    when A_1 =>  -- implements mode 1 for Port A
     …
    when A_2 =>  -- implements mode 2 for Port A
     …
end case;
```

If *a_mode* is a constant, then two of the when statements will never be executed and can therefore be eliminated. Because these statements implement much of the functionality of the port, large area savings can be achieved.

5.2 8237A DMA Controller

The 8237A had several configuration registers, including those that select the arbitration mode, the number of active channels, and the transfer mode, base address, and block size of each channel. We examined the situation of using only a single channel, in single transfer mode. This reduced the model from 920 to 435 lines.

The following code shows the channel being selected based on the configuration register, *command*, and the input *drequest*:

```
if ( command(4) = '0' ) then
     if ( drequest(0) = '1' ) then
          channel <= 0;
```

```
        elsif ( drequest(1) = '1' ) then
             channel <= 1;
        elsif ( drequest(2) = '1' ) then
             channel <= 2;
        elsif ( drequest(3) = '1' ) then
             channel <= 3;
        end if;
```

If the program writes a constant to *command* and *dreqeust*, this implies that only a single channel is being used. The *channel* signal is used frequently in the code, as shown below:

```
   db   <= curr_addr(channel)(15 downto 8);
   a7_4 <= curr_addr(channel)(7 downto 4);
   a3_0 <= curr_addr(channel)(3 downto 0);
   dack(channel) <= command(7);
```

This code can be further optimized by propagating the constant value of *channel*. *Curr_addr* is implemented as 4 separate 16-bit registers, one for each channel. Therefore, if we are only using one channel, we can completely remove the three other registers. This optimization also applies to other registers in the design, such as those that store the base address for each channel. In the entire system, fifteen different 16-bit registers can be eliminated by this one constant.

5.3 PC16550A UART

The PC16550 also had several configuration registers, including those that enable transmit and receive, select the interrupt mode, and select the baud rate. We examined two situations, one where the device was configured for transmit only at a specific baud rate, and the other where it was configured for receive only at a specific baud rate. Each reduced the model's lines of code from 1063 to roughly 625.

Configuring the UART for transmit only allows for the entire process associated with receiving to be removed from the code. By making the baud rate a constant value, all registers used to store the baud rate and all logic required to read the baud rate from the input are removed. In addition, a custom counter can be implemented in order to generate the fixed baud rate.

5.4 Free-DCT-L Core

The DCT core had configuration registers for selecting between forward and inverse DCT, and for selecting among 8, 9, 10, or 12-bit resolution. The basic structure of the DCT core is shown *Figure 5*. Note that the controller and cyclic register components both have configuration signals as input, making them obvious candidates for constant propagation.

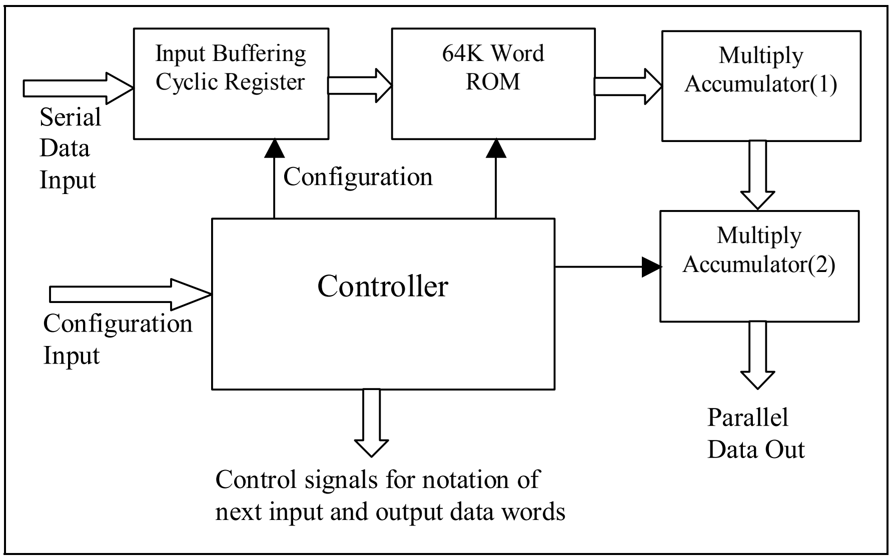


Figure 5. Block diagram of DCT core.

Portions of the VHDL code for the controller that deal with control registers are shown below:

```
with dctselect select
     rows <= add_tmp2(5 downto 3) when '1',
             add_tmp2(2 downto 0) when others;

with dctselect select
     columns <= add_tmp2(2 downto 0) when '1',
                add_tmp2(5 downto 3) when others;
```

The signal *dctselect* is used to select between inverse and forward DCT. By converting *dctselect* into a constant with a value of zero, we can optimize the code into the following assignments:

```
rows    <= add_tmp2(2 downto 0);
```

```
  columns <= add_tmp2(5 downto 3);
```

This change eliminates two multiplexors (used to select between *add_tmp2*(5 downto 3) and *add_tmp2*(2 downto 0)) and a small amount of wires.

The controller also contains a finite state machine that depends on the control registers. The following code represents a state from the FSM that reads the *mode* signal in order to select the resolution:

```
  when 5 =>
     if mode = "00" then
         state := 6;
     elsif mode = "01" then
         state := 12;
     elsif mode = "10" then
         state := 11;
     else
         state := 9;
     end if;
```

By converting *mode* to a constant, we can convert this code into a single assignment to the state variable. This eliminates the need for the 3 comparisons. In addition, several of the states can only be reached from the assignments in this control statement. Therefore, depending on the value of the constant assigned to *mode*, we may be able to eliminate states from the FSM. This type of optimization might be very difficult for a normal compiler to make because it would have to prove that a certain region of code could not be reached. However, for a synthesis tool, constant propagation can be followed by FSM state minimization. It would not be hard to detect states that were never reached because there would be no transitions to these states. Therefore, a synthesis tool could easily remove unneeded states resulting from constant propagation.

The cyclic register component also uses the same control register, *mode*, which selects the resolution. The code is shown below:

```
if rising_edge(ck) then
  if rst = '1' then
    internal <= (others => '0');

  -- 8 bits resolution mode
  else mode = "00" then
     internal(63 downto 1) <= internal(62 downto 0);
```

```
      internal(0) <= din_tmp;

   -- 9 bits resolution mode
   elsif mode = "01" then
      internal(71 downto 1) <= internal(70 downto 0);
      internal(0) <= din_tmp;

   -- 10 bits resolution mode
   elsif mode = "10" then
      internal(79 downto 1) <= internal(78 downto 0);
      internal(0) <= din_tmp;

   -- 12 bits resolution mode
   else
       internal(95 downto 1) <= internal(94 downto 0);
       internal(0) <= din_tmp;
   end if;
end if;
```

In this example, *internal* is likely to be synthesized as a 96-bit shift register, where shifts only occur in specified ranges. By assigning *mode* a constant value of "11", the code is simplified by requiring only one shifting range of 95 bits, resulting in much smaller hardware.

In our experiments, we tested the configuration of inverse DCT with 12-bit resolution. This configuration reduced the size of the code from 910 lines to 867 lines.

5.5 Results

Note that the parameters used in all the described examples were not represented by constants or generics in the VHDL source. Rather, the cores were designed to be synthesized to support software configuration of these parameters, as is common.

The size and power data is summarized in *Table 1*. We see that size after synthesis was reduced by an average of 58%, and power by an average of 22%. The reason that power is not reduced as much as size is because many of the gates eliminated through constant propagation were not used during a core's execution even when present, so didn't consume much power. This can be seen in the UART example, by the removal of logic to handle receiving. This logic might be a large part of overall area, but if it isn't being used frequently, it is unlikely to consume much power. We believe that the power reductions that do occur result from less switching activity due to simpler control and datapath switching logic. For the DCT core, power savings are actually greater than area savings. This results from the removal of a small amount of frequently used logic that contributes greatly to the switching activity of the system.

Table 1. Comparison of cores before and after constant propagation.

Cores	Gates, Original	Gates, with constant propagation	% size savings	Power, original (micro-watts)	Power, with constant propagation (micro-watts)	% power savings
8225A mode-0	3069	834	73%	2772	1902	31%
8225A mode-1	3069	918	70%	2915	2098	28%
8225A mode-2	3069	953	69%	2952	2124	28%
8237A single transfer	7276	2344	68%	2453	2097	15%
PC16550 Tx	2503	1169	53%	1461	1249	15%
PC 16550 Rx	2503	1188	53%	1449	1307	10%
DCT Forward 8-bit	2295	1872	18%	1391	958	31%
Average savings			**58%**			**22%**

These reductions come of course at the cost of not being able to reprogram the configurable parameters of the core once the system has been implemented. Thus, if modifying the microprocessor's program is a possibility, then propagating constants across the software/hardware boundary should either not be done, or should be done only to the extent that the designer is certain that particular constants won't change. However, as mentioned earlier, many embedded systems have their programs fixed in mask-programmed ROM, and thus the configurable parameters could never have been modified anyway, meaning our approach would have no impact on flexibility in those cases.

6. FUTURE WORK

A core-based design flow often involves more than a microprocessor and peripheral cores. In many cases, co-processors or custom hardware may be used in order to speed up frequent operations. A common example of this is adding a floating-point co-processor core to a microprocessor core with only an integer pipeline. We have previously studied the effects of moving frequently executed loops into custom hardware cores, where we are more concerned with improving energy efficiency as opposed to reducing area.

In some cases, we can apply constant propagation optimizations to these custom cores. For example, consider a core that implements a frequently executed function that takes several parameters as input and returns a value based on the inputs. If we trace the values of the inputs and can determine that one of the parameters has the same value a large percentage of the time, we can create a custom hardware implementation of the function that treats the parameter as a constant. This allows us to perform all optimizations associated with constant propagation, resulting in a smaller, faster, and more energy efficient core. Of course, since we are implementing the function for only one value of the inputs, we would either have to create an additional core to implement the function for all other inputs, or simply execute the function in software. The latter method would have the additional overhead of checking the values of the function in software in order to determine whether the function should be executed in the optimized hardware. We plan to investigate the benefits of such approaches in the future.

7. CONCLUSIONS

As core-based design methodologies grow in popularity, cores will be heavily parameterized to increase applicability and hence sales. Pre-fabrication parameters, specified using HDL generics or constants, can result in optimized hardware. However, post-fabrication parameters, known as software configurable parameters, until now have not been exploited similarly. We introduced the idea of propagating constants beyond the microprocessor software, to the peripheral hardware. We showed that such propagation yielded reductions in size by 58%, and good power reductions of between 10-30%, using several standard peripheral examples. This work is part of the UCR Dalton project, which seeks to develop techniques for parameterized core-based system-on-a-chip design [7]. This work motivates the need for future work on system-on-a-chip frameworks whose compilers are able to detect "constants" in the sense of software configurable register values, and are able to coordinate between compilers and synthesis tools to propagate those constants to hardware.

ACKNOWLEDGMENTS

This work was supported by the National Science Foundation under grant number CCR-9876006. We would like to thank Rilesh Patel for his contribution to this work.

REFERENCES

[1] Aho, A.V., R. Sethi, J.D. Ullman. "Compilers: Principles Techniques, and Tools," Reading, Addison-Wesley Publishing Company, March 1998.

[2] Escalade Corporation, http://www.escalade.com/.

[3] Gupta, R., and Y. Zorian. Introducing Core-Based System Design. IEEE Design & Test, Vol. 14, No. 4, Oct-Dec 1997, pp. 15-25.

[4] Inventra core library, Mentor Graphics, http://www.mentor.com/inventra/.

[5] Platform Express. Mentor Graphics, http://www.mentor.com/soc/platform_ex/.

[6] Stitt, G., F. Vahid, T. Givargis, and R. Lysecky. A First-step Towards an Architecture Tuning Methodology for Low Power. Compilers, Architectures, and Synthesis for Embedded Systems (CASES'00), November 2000, pp. 187-192.

[7] The UCR Dalton project: http://www.cs.ucr.edu/~dalton.

[8] Vahid, F., and T. Givargis. Platform Tuning for Embedded Systems Design. IEEE Computer, Vol. 34, No. 3, March 2001, pp. 112-114.

[9] Virtual Socket Interface Association, Architecture Document, http://www.vsi.org, 1997.

[10] Wegman, M., and F.K. Zadeck. Constant Propagation with Conditional Branches. ACM Transactions on Programming Languages and Systems, Vol 18, No 2, April 1991, pp. 181-210.