A Framework for Bitwidth Optimization in System-on-Chip Design

Tomiyama, Hiroyuki
Institute of Systems & Information Technologies/KYUSHU

Cao, Yun
System LSI Research Center, Kyushu University

Mesbah, Uddin
Department of Computer Science and Communication Engineering, Kyushu University

Inoue, Akihiko
Matsushita Electric Industorial Co., Ltd.

他

A Framework for Bitwidth Optimization in System-on-Chip Design

Hiroyuki Tomiyama *1 Yun Cao *2 Uddin Mesbah *3 Akihiko Inoue *4
Eko Fajar *5 Hajime Yamashita *6 Hiroto Yasuura *2*3

*1 Institute of Systems & Information Technologies/KYUSHU
*2 System LSI Research Center, Kyushu University
*3 Department of Computer Science and Communication Engineering, Kyushu University
*4 Matsushita Electric Industorial Co., Ltd. *5 Sony LSI Design Inc. Kyushu *6 Hitachi, Ltd.

Abstract

Advances in semiconductor technology and increased reusability of IPs enable embedded system designers to develop Systems-On-Chip (SOCs) which are customized for a specific application (or application domain). Specifically, customization of processor architecture and memories is enabled, which allows efficient implementation of SOCs. Up to now, a number of research efforts on processor customization have been made in both industry and academia, where the instruction set and the datapath configuration are prime design parameters. In this paper we demonstrate that the datapath bitwidth has a large impact on system performance, chip area, and energy consumption, and then propose a framework and tools for datapath bitwidth optimization. Two case studies are presented to prove the effectiveness of the optimization.

1. Introduction

Traditional embedded systems are implemented on a system board where several chips (e.g., a processor, memories, ASICs, and peripherals) are connected to each other. A designer selects a processor and memories from standard product libraries, and designs ASICs if necessary. In such a design flow, the designer is not allowed to customize the processor and memories. However, advances in semiconductor technology and increased reusability of hard and soft IPs enable embedded system designers to develop Systems-On-Chip (SOCs) which are highly customized for a specific application (or application domain). Specifically, customization of the processor architecture and the memory system is enabled, which permits efficient implementation of SOCs in terms of performance, chip area and power/energy consumption. A number of research efforts on processor customization have been made in both industry and academia so far, where the instruction set and the datapath configuration are prime design parameters.

In this paper, we demonstrate that datapath bitwidth of SOCs significantly affects the performance, cost, and power/energy consumption. In general, a large datapath bitwidth increases the cost and power of datapath components (such as functional units, registers, data memories, and so on). The clock frequency also becomes worse. On the other hand, a small datapath bitwidth requires a large number of instructions to be stored in program memory and be executed, resulting in low performance, high memory cost, and high energy. Since each application has its own design goal and constraints, optimization of the datapath bitwidth for each application is very effective in SOC design [4, 5, 6]. By changing the datapath bitwidth, we can optimize performance, cost, and power/energy consumption. To enable the bitwidth optimization, of course, a new design methodology and several design support tools are necessary.

In this paper, we present a framework, tools and techniques for designing processor-based SOCs, with specifically focusing on the datapath bitwidth optimization. First, Section 2 shows why we focus on the datapath bitwidth optimization. Section 3 presents an overall framework for designing SOCs with optimized datapath bitwidth. Section 4 describes a configurable processor architecture, named soft-core processor. Next, Sections 5 and 6 present the Valen-C language and a retargetable Valen-C compiler. Then, techniques to analyze computational precision required by application programs are described in Section 7. Section 8 presents two case studies on datapath bitwidth optimization. Finally, Section 9 concludes this paper with a summary.

2. Datapath Bitwidth as a Parameter in SOC Design

This section shows that the datapath bitwidth significantly affects the performance, cost, and energy

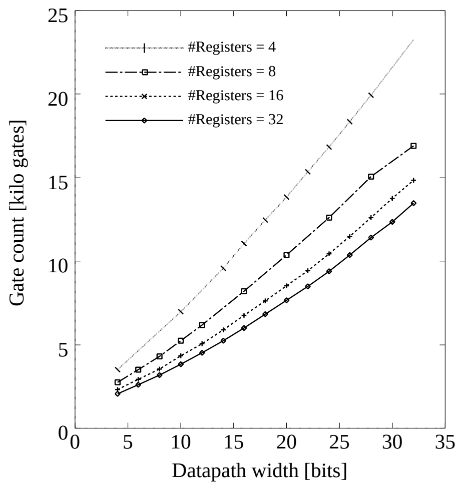


Figure 1. Processor cost versus datapath widths

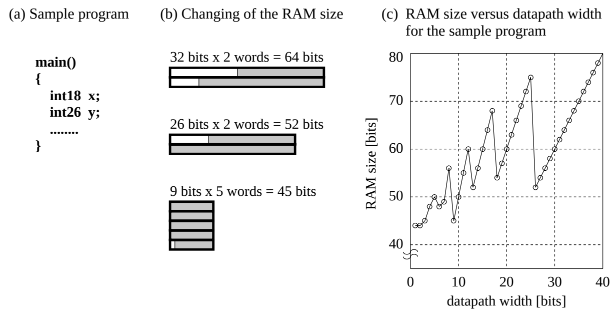


Figure 2. Data memory size for various datapath widths

consumption of processor-based SOCs.

2.1. CPU Area and Performance

We have designed a processor whose datapath width can be changed by designers [8]. As the datapath width is reduced, the processor cost almost linearly decreases. As indicated in Figure 1, the CPU cost is reduced by about 60% when the datapath width is reduced from 32 bits to 8 bits. Generally, narrowing the datapath width reduces processor cost, but degrades the performance. This is because the number of execution cycles increases since some of single precision operations become double or more precision ones. Single precision operations mean ones whose precision is smaller than the datapath width. For example, an addition of two 32-bit data is a single precision operation on processors whose datapath width is equal to or greater than 32 bits, while it is a double precision operation on a 16-bit processor.

It should be noted that the clock frequency can be enhanced when narrowing the datapath width. In our experience, however, the clock frequency improvement rarely pays for the increase in the number of instructions. We can conclude that in most cases narrowing datapath width degrades the processor performance.

2.2. Data RAM Area

Changing the datapath width affects the size of a data memory (RAM) and an instruction memory (ROM) as well as processor cost [5].

Let us consider a program including two variables as shown in Figure 2, and assume that two variables x and y require at most 18 bits and 26 bits, respectively. When the datapath width is 32 bits, two words are required to store the two variables. Therefore, the amount of the data memory is 64 bits. Since the minimum size required to store them is only 44 bits (= 18 + 26), 20 bits of them (about 30%) are unused. When the datapath width is 9 bits, two words and three words are required for x and y, respectively, and the unused area is only 1 bit. Figure 2 (c) shows that the RAM size does not decrease monotonically with the reduction of the datapath width. Quite many unused bits can be eliminated by determining the datapath width appropriately.

2.3. Instruction ROM Area

The ROM size in bits is calculated by multiplying the instruction word length by the number of instructions stored in the ROM. When the datapath width is reduced, the number of instructions in the ROM increases. For example, an addition of 20-bit data can be executed by only one instruction on a 20-bit processor. However, if the datapath width is 10 bits, at least two instructions are necessary. Furthermore, extra load and store instructions may be required because of the shortage of registers. The ROM cost grows monotonically as narrower the datapath width is, if the instruction word length does not change.

2.4. Energy Consumption

Energy consumption changes in a non-monotonic manner with an increase in datapath bitwidth. In general, if the datapath is too narrow, energy is increased because of the increased execution cycles. On the contrary, if the datapath is too large, energy is also increased due to the increased number of gates to be switched.

2.5. Examples of Datapath Bitwidth Exploration

In the rest of this section, we present three examples of datapath bitwidth exploration in the design of appli-

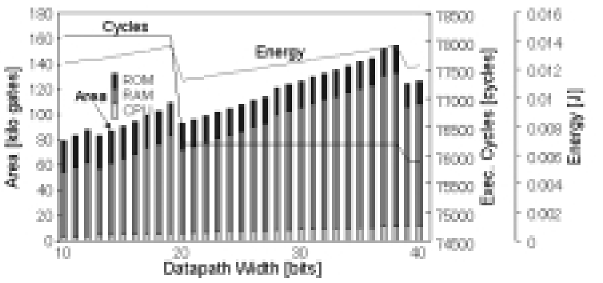


Figure 3. An example of datapath bitwidth exploration for a calculator

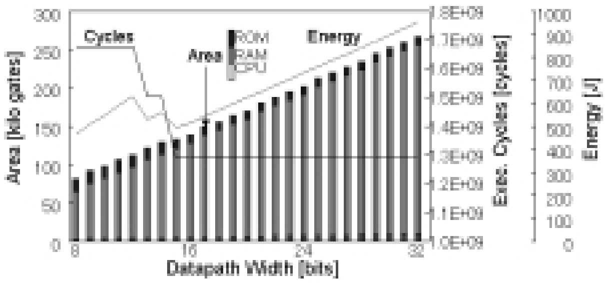


Figure 4. An example of datapath bitwidth exploration for a Lempel-Ziv encoder/decoder

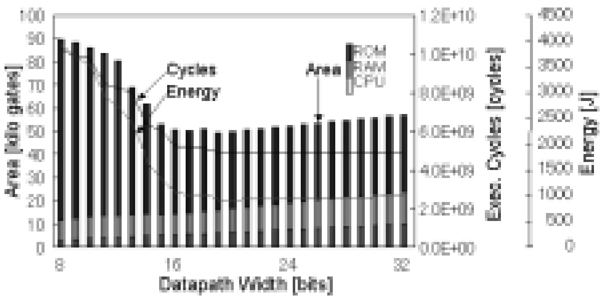


Figure 5. An example of datapath bitwidth exploration for an ADPCM encoder

cation specific processors. The following three applications are used: a 12-digit decimal calculator, a Lempel-Ziv encoder/decoder, and an ADPCM encoder. For each of the three applications, we estimated performance (in terms of execution cycles), chip area (including CPU, RAM and ROM), and energy consumption while varying datapath bitwidth. The estimation results are described in Figures 3, 4 and 5.

We can see that performance, cost and energy largely depend on the datapath bitwidth. From the estimation results, we can find the optimal solution of the bitwidth. Note that each application has different characteristics on performance, area, and energy. Thus, the optimal bitwidth varies depending on the application.

3. A Framework for Bitwidth Optimization

Figure 6 shows our framework for designing SOCs in which the datapath bitwidth is a design parameter. The methodology starts from an application program written in the ANSI-C language and a template processor core, called *soft-core processor*, whose datapath is 32 bits. The datapath bitwidth can be optimized in the later design phases. In order to find the optimal the datapath bitwidth, we need to analyze the application program. This process is called *variable size analysis*. In the variable size analysis process, the required precision (bitwidth) for each variable in the application program is analyzed statically or dynamically. Then, the bitwidth information needs to be annotated in the program. Since ANSI-C does not support specification of the bitwidth, we extended the ANSI-C language to enable it. The new language is called *Valen-C*. After translation of the C program into a Valen-C program, HW/SW partitioning and processor customization are performed. After these processes, we obtain a hardware description in HDL or Valen-C, a software description in Valen-C, and a processor-memory configuration description. Finally, those descriptions are synthesized or compiled to obtain an SOC design.

4. Soft-Core Processor

Traditionally, processor cores have been used as hard macros. An SOC designer can copy a layout data of a processor core into his/her design, but cannot modify its architecture to fit to application. We call such a processor core *a hard-core processor*. On the contrary, we call a configurable processor core *a soft-core processor*. Soft-core processors have several parameters to be customized by SOC designers for each application. The typical parameters include the number of registers, the word length of data and/or instructions, the number of operation units and so on. The soft-core processor may be presented in the forms of a fabricated chip, layout data, net list in logic circuit level and RTL description in HDL. Design modification is done mainly in the HDL description rather than net list or layout levels. A customized processor is obtained through the redesign process using some synthesis tools. Options or scripts for the synthesis tools are also provided for prompt re-synthesis after the modification.

We have designed an example of soft-core processor, named Bung-DLX [8]. Bung-DLX is based on the DLX architecture [3]. In the design of the soft-core processor architecture, the following requirements have to be con-

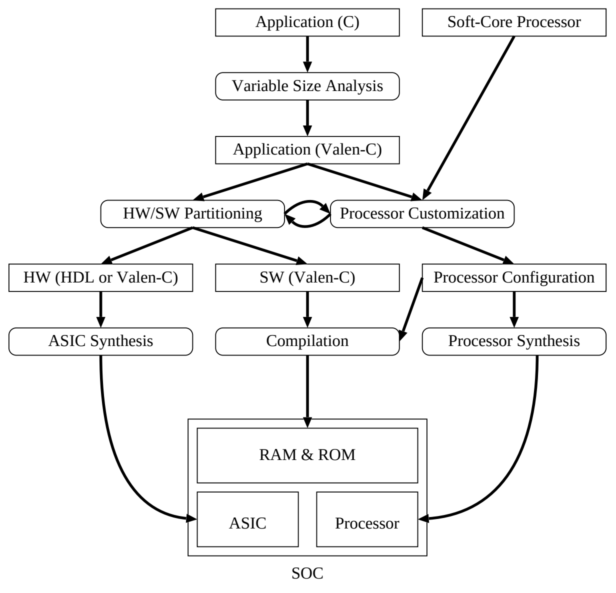


Figure 6. A Framework for Bitwidth Optimization

sidered. 1) Simple and clear architecture for easy modification: For the modification in the redesign phase, the architecture of the soft-core processor should be simple and easy to understand for system designers. The architecture affects on the configuration of the design environment. In Bung-DLX, a simple RISC architecture based on well-known DLX architecture is adopted. The modification in the word length of datapath, the number of registers, the amount of memory space and the instruction set can be done by changing design parameters in a design modification table. In other words, the modifiable parameters are restricted by the design modification table. The HDL description of the modified Bung-DLX is automatically generated from the design modification table. Since Bung-DLX has the Harvard architecture, in which instruction memory is separated from data memory, the word length of datapath (i.e. the width of data memory) can be changed independently from the length of instruction words. 2) Simple interface between the processor and software: In the partitioning of hardware and software, we have to redesign the interface between software and hardware. The original soft-core processor should have a simple interface in order to make compiler generation and modification of software library easy. Since RISC architecture provides a simple interface between processors and software and the modification of the architecture of Bung-DLX is restricted, most problems in software interface caused by modification can be solved in a compiler. A retargetable compiler, called the Valen-C compiler, was developed and all application programs designed on the original architecture can be automatically compiled to the modified architecture. The information of the modified architecture is provided from the design modification table and automatically reflected in the architecture definition for the retargetable compiler. 3) Achieving high performance and/or low cost implementation: The soft-core processor should compete with hard-core processors in the performance, chip area, power consumption and implementation cost. This requirement has still not been solved completely. Many possibilities to improve performance and cost of Bung-DLX are remained for the future work.

The original Bung-DLX has a non-pipelined RISC-type architecture with 32 general purpose registers and 72 instructions. The length of data and instruction words is 32 bits. The address spaces of data memory and instruction memory are both 2^{32}. It is described by a VHDL code with about 7,000 lines. The gate size after logic synthesis is 23,282 gates. The design modification table includes the word length of data path, the amount of data memory, the length of instruction word, the amount of data memory, the number of registers and the instruction set. Bung-DLX is now provided in the form of a VHDL description together with a simulator, an assembler, and a compiler.

5. The Valen-C Language

Currently, the ANSI-C language is widely used as a programming language for designing embedded software. In ANSI-C, semantics of programs depend on both processor architectures and compilers. The size of each data type is determined by compilers. Typically, the size of *char* is 8 bits, *short* is 16 bits, *int* is 32 bits, and *long* is also 32 bits. Some recent compilers support the *long long* type of 64 bits. Note that the sizes may vary in different compilers. Therefore, the portability of C programs is limited. For example, C programs written for a 32-bit processor may not run correctly on 16-bit or 24-bit processors.

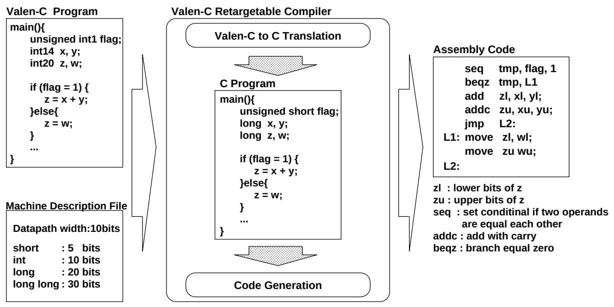
In order to overcome the drawback, we have developed an extended C language, called Valen-C (Variable Length C) [8]. Valen-C enables system designers to explicitly specify the required bit length of each variable in programs. Even if system designers customize the datapath width for their application, the Valen-C compiler preserves the semantics of the program. Therefore, Valen-C programs can be reused on processors with various datapath widths. Valen-C is one solution for the problem of word-length support in C.

Syntactically, Valen-C is an extension of the C language. As mentioned before, in Valen-C, programmers can specify the required bit length of each variable in a program. The control structures in Valen-C, such as "if" and "while" statements, are same as C.

C provides for three integer sizes, declared using the keywords *short, int* and *long*. The sizes of these integer types are determined by the compiler designer. In many processors, the size of *short* is 16 bits, *int* is

16 or 32 bits, and *long* is 32 bits. On the other hand, in Valen-C, programmers can use more kinds of data types. For example, if a variable x needs a precision of 11 bits, x will be declared as "*int11 x*". Similar to C, the *sign* and *unsign* qualifiers can be specified in Valen-C. The *char* type also exists in Valen-C, and it is assumed to have a length of larger than 8 bits. The struct type and the array type are also available as well.

A floating point variable which has the precision of a 5-bit exponent and a 10-bit mantissa is declared as "*float5.10 x*".



```
main(){
      unsigned int1 flag;
      int14  x, y;
      int20  z, w;

      if (flag = 1) {
          z = x + y;
      }else{
          z = w;
      }
      ...
}
```

```
Datapath width:10bits

short      : 5   bits
int        : 10 bits
long       : 20 bits
long long  : 30 bits
```

```
main(){
      unsigned short flag;
      long  x, y;
      long  z, w;

      if (flag = 1) {
          z = x + y;
      }else{
          z = w;
      }
      ...
}
```

```
        seq     tmp, flag, 1
        beqz    tmp, L1
        add     zl, xl, yl;
        addc    zu, xu, yu;
        jmp     L2:
  L1:   move    zl, wl;
        move    zu wu;
  L2:
```

```
zl  : lower bits of z
zu : upper bits of z
seq  : set conditinal if two operands
         are equal each other
addc : add with carry
beqz : branch equal zero
```

Figure 7. Example of compilation of Valen-C program

6. Retargetable Valen-C Compiler

If a processor architecture is modified, the compiler for the processor also need to be modified. To make the modification easy, retargetable compilers have been developed so far by some researchers and engineers. However, most of these compilers assume that the datapath width is 8×2^n (n is a natural number). In our optimization, a retargetable compiler which is applicable to any datapath width is required. In cases that the precision of an operation is larger than the datapath width, the compiler has to translate the operation into a certain number of machine instructions.

We have developed such a retargetable compiler for the Valen-C language [8]. The Valen-C compiler takes a Valen-C or ANSI-C program as input and generates assembly code.

The Valen-C compiler preserves the precision of programs in the following manner: If a variable has a precision of n bits, the Valen-C compiler allocates the storage of not less than n bits for the variable. If an operation in a Valen-C program requires the precision of n bits, the operation is performed with the precision of not less than n bits. For example, an addition of two 13-bit variables will be calculated with a precision of 20 bits on 20-bit processors. In cases that the precision of an operation is larger than the datapath width, the operation is performed by more than one machine instructions. For example, an addition with a 20-bit precision is performed by two addition instructions of lower 10 bits and upper 10 bits on a 10-bit processor. Floating point data types have not been supported yet.

The Valen-C compiler is retargetable by modifying the machine description. The machine description includes the datapath width, the number of registers, the instruction set, the sizes and alignments of the program and data memories, the minimum addressable size of the data memory, and so on.

The datapath width of the processor does not have to be 2^n (n is a positive integer number). For example, the datapath width can be 11 bits or 29 bits. Furthermore, variables whose sizes are larger than the datapath width is available in Valen-C programs.

Figure 7 shows an example of the compilation of a Valen-C program on a 10-bit processor. The example assumes that the size of *short, int, long*, and *long long* is 5 bits, 10 bits, 20 bits, and 30 bits, respectively. In the Valen-C to C translations phase, the variable *flag* is mapped into *short* type since the size is less than 5 bits. The *long* type is used for x, y, z and w because their sizes are larger than 10 bits and do not exceed 20 bits. In the code generation phase, the equation $z = x + y$ is performed by two addition instructions of lower 10 bits and then upper 10 bits with a carry bit. Similarly, the assignment $z = w$ is also divided into two machine instructions.

The Valen-C compiler has been implemented based upon the SUIF library [2], and is available at [1].

7. Variable Size Analysis

Although the Valen-C language has enhanced reusability over ANSI-C, specifying bitwidth of every program variable is very cumbersome and time-consuming. Therefore, many system designers may prefer ANSI-C rather than Valen-C. As mentioned earlier, however, ANSI-C cannot be used for datapath bitwidth optimization in SOC design.

In order to reduce the burden of Valen-C programmers, we have developed techniques for variable size analysis which automatically analyze required bitwidth of variables in C programs [7]. Using the techniques, C programs are automatically translated into Valen-C ones. Thus, programmers do not have to care about variable bitwidths.

There exist two approaches to analyze variable sizes. One is dynamic analysis which executes programs and monitors the value of each variable. The other approach is static analysis, which analyzes variable sizes without running programs. The rest of this section presents these approaches.

7.1. Static Approach

For an assignment statement with arithmetic operations, the range of a variable in the left side is calculated from ranges of variables, constants, and operators in the right side.

Let us consider the following assignment statement with a single operator.

$y = x * 3;$

In this example, if a variable x is in $[-2, 5]$, the range of y, $[-6, 15]$, is analyzed. Therefore, the size of y is 5 bits.

Let us consider another example with more than one operators.

$z = x * 3 + y;$

In order to calculate the range of z, the range of $x * 3$ is firstly calculated. Then, we can obtain the maximum value of z by adding the maximum value of $x * 3$ to that of y. The minimum value of z can be calculated by adding the minimum value of $x * 3$ to that of y.

Next, let us consider the case where more than one assignments exist in a function. The variable ranges in the function can be calculated by applying above analysis methods to each assignment successively from the top of the function. If a variable appears in the left side in more than one assignments, the maximum value of the results each of which is calculated in each assignment becomes the maximum value of the variable. The minimum value can be obtained in a similar way.

The following example shows a function which has more than one assignments.

```
int func(int x)
{
    int y,z;
    z = x * 3;          — (1)
    y = z + 3;          — (2)
    z = x * 2 + y;      — (3)
}
```

The function *func*() has three assignments (1), (2), and (3). In this example, assume that the range of the parameter x is $[-1, 2]$. The sizes of local variables y and z are calculated in the following manner. First, the assignment (1) is analyzed and we can know that the range of z is $[-3, 6]$. Next, by analyzing the assignment (2), we obtain the range $[0, 9]$ of y. And next, after analyzing the statement (3), we can know that the range of z at the point (3) is $[-2, 13]$. Finally, the results of analyzing assignments (1) and (3) are merged and the range $[-3, 13]$ of z is obtained. Then, it is analyzed that the sizes of y and z are both 5 bits.

For logical operations, it is difficult to calculate maximum and minimum values of each variable. We compute the variable size for logical operations without calculating the maximum and minimum values. Let us consider the following assignment with a logical operation and assume that the range of y is $[0, 5]$.

```
unsigned int x, y;
x = y & (unsigned)4;
```

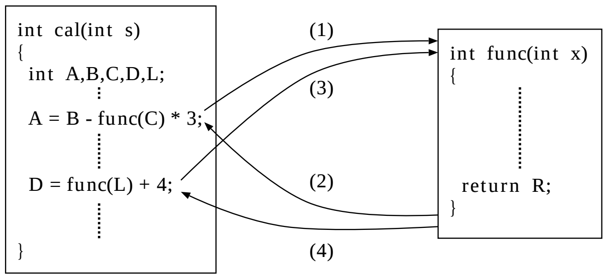


Figure 8. Function calls

In this example, since sizes of both y and the unsigned constant 4 are 3 bits, we can know that the size of x is also 3 bits.

Conditional Statements: For a conditional statement, the *then* and *else* parts are analyzed separately. Let us consider the following *if-else* statement where the ranges of y and z are assumed to be $[3, 6]$ and $[-13, -10]$, respectively.

```
if  (a < 5)  x = y − 5;
      else   x = z + 13;
```

The range of x, $[-2, 1]$, is obtained in the *then* part, and $[0, 3]$ in the *else* part. Then, the range $[-2, 3]$ is obtained for x.

Loops: We consider two types of loops. One is bounded loops, and the other is unbounded loops. A bounded loop is expanded into a straight-line program. Then, variable sizes in the loop can be obtained. On the other hand, it is impossible to expand unbounded loops because the number of iterations can not be known statically. In this case, dynamic analysis is applicable.

Arrays: Let us consider how to analyze the size of arrays. Basically, each element of an array is treated as a scalar variable and the analysis methods explained above are applied. Then, the size of the array is given by the largest size of the elements. In case that it is impossible to analyze the value of the array index, all the array elements are considered as one scalar variable.

Function Calls: Let us consider the example in Figure 8 where a function *cal*() calls *func*() twice. When analyzing the assignment to A, *func*() is analyzed with the range of the parameter C which is assigned to x in *func*(). Then, we get the range of R which is used for analyzing the range of A in *cal*(). When analyzing the assignment to D, *func*() is analyzed again by assuming that the range of x is the same as that of L. Finally, the range of parameter x in *func*() is analyzed by merging the ranges of C and L. Since recursive function calls generally cannot be analyzed statically, we use dynamic analysis for them.

Pointers: The size of pointers is the same as the memory address width and is determined by hardware organization of the system.

Pointer Accesses: Let us consider the case that variables are accessed via pointers. In general, it is not always possible to analyze which variable a pointer p pointing to. If possible, we can use methods presented above. If otherwise, first, we analyze the set of variables to which p can pointed. In case that a reference $*p$ appears in the right part of assignments, the maximum and minimum values of $*p$ are assumed to be maximum and minimum values of the variables in the set. In case that $*p$ appears in the left size of assignments, the ranges of all the variable in the set are updated.

7.2 Dynamic Approach

Static analysis is an efficient method to analyse the variable size. However, in many cases when we can not predict the assigned value of a variable unless we execute the program, such as the case of unbounded loops, static analysis becomes insufficient. As a solution to this problem, we adopted dynamic analysis.

In dynamic analysis, we execute the program and monitor the values assigned to each variable. For this purpose, after each assignment statement in program code, a function is inserted which monitors the variable in the assignment statement. The monitoring function checks the value assigned to the variable and verifies the bit width required to store it. After that, it keeps the bit width temporarily in a table. Next, when the monitoring function checks the same variable with a different assigned value, it compares the new bit width with the bit width already memorized in the table, and keeps the bigger bit width in the table, and so on. Thus, the required bit width of the variable is analized while running the program.

8. Case Studies

We have performed a number of case studies on datapath bitwidth optimization. This section presents two of them, i.e., the design of ADPCM decoder LSIs and the design of MPEG-2 video decoder processors.

8.1 ADPCM Decoder LSIs

We have designed two ASICs for ADPCM decoder. The design started from an ADPCM decoder program written in C which is a part of the DSPstone benchmark suite [9]. Next, we statically analyzed required bitwidth of variables in the program. The analysis results are shown in Table 1. There are eight *int*-type variables in the program which are all 32 bits in original. Our variable size analysis results show that no variable requires the precision of 32 bits or more. The size of the largest variable is only 18 bits.

Table 1. Variable size analysis results for ADPCM decoder

Variables	Original bitwidth	Required bitwidth
valpred	32 bits	18 bits
index	32 bits	9 bits
step	32 bits	15 bits
bufferstep	32 bits	1 bit
delta	32 bits	4 bits
sign	32 bits	4 bits
vpdiff	32 bits	16 bits
inputbuffer	32 bits	8 bits
Total	256 bits	75 bits (−71%)

Table 2. Synthesis results of ADPCM decoders with different datapath bitwidth

	ADPCM32	ADPCM18	
Bitwidth	32	18	
Energy [nJ]	365.58	238.76	(-35%)
# Cells	1379	669	(-52%)
# Tr.	13006	5864	(-55%)

Based on the results, we designed two ASICs for ADPCM decoder. One has 32-bit datapath, and the other has 18-bit one. At that time, no high-level synthesis tool for C/Valen-C was available, we manually designed the ASICs in VHDL. Then, logic synthesis was performed with Synopsys Design Compiler and 0.5 μm standard cell technology. The synthesis results are summarized in Table 2. With datapath bitwidth optimization, chip area and energy concumption were significantly reduced (by 49% and 35%, respectively).

8.2 MPEG-2 Video Decoder Processors

In the second case study, we have examined the performance/cost/energy trade-off by changing datapath bitwidth of a soft-core processor. MPEG-2 video decoder was used as an application program. The original program consists of over 6,000 lines of C code which includes several function blocks such as IDCT blocks, a couple of motion estimation blocks, a motion compensation block, variable length encoding, decoding blocks and so on.

We have analyzed required bitwidth of 384 *int* type variables, and the results are summarized in Table 3. Based on the results, we translated the C program into Valen-C one. For variables of the other type (e.g., *char*,

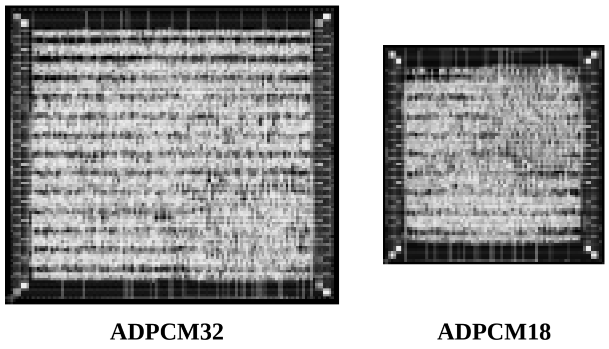


Figure 9. Layout of the ADPCM LSIs

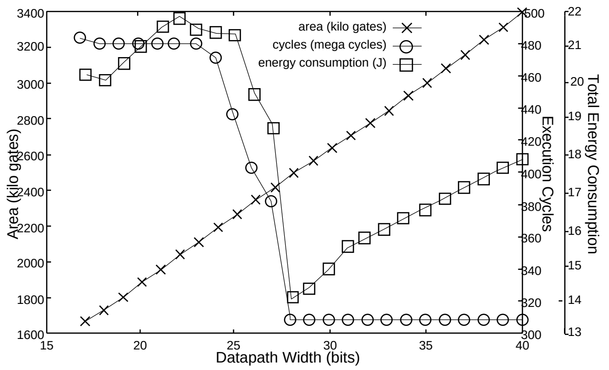


Figure 10. Performance/cost/energy trade-off for MPEG-2 decoder processor

Table 3. Variable size analysis results for MPEG-2 decoder

Size	# Variables	Size	# Variables
1 bit	50	15 bits	2
2 bits	17	16 bits	39
3 bits	11	17 bits	39
4 bits	11	18 bits	3
5 bits	10	20 bits	6
6 bits	14	24 bits	14
7 bits	16	26 bits	2
8 bits	9	27 bits	4
9 bits	7	28 bits	3
10 bits	3	29 bits	3
11 bits	6	30 bits	7
12 bits	17	32 bits	82
14 bits	46		

short, etc.), their bitwidth was kept same.

We varied the datapath bitwidth of the processor from 17 bits to 40 bits, and estimated the performance (in terms of execution cycles), cost (gate count) and energy consumption. The results are depicted in Figure 10. From the figure, we can see that the chip area increases in a monotonic fashion with the datapath bitwidth. Execution cycles are minimized at 28-bit datapath, but are not further decreased for larger bitwidth. Please note that, in general, smaller datapath bitwidths have shorter critical path delays. This means that, in the MPEG-2 example, performance is maximized at 28-bit datapath. Energy consumption is also minimized at 28 bits. For datapath shorter than 28 bits, more energy is required because of larger execution cycles. On the other hand, for datapath larger than 28 bits, more gates need to switch, thus more energy is consumed.

Thus, we can explore the design space to find the optimal cost/performance/energy trade-off by changing processor datapath bitwidth.

9. Conclusions

We have presented an overall framework, tools, and techniques for datapath bitwidth optimization in the design of SOCs. Through the case studies, we have demonstrated that an SOC design can optimize the performance, cost, and energy trade-off by changing the datapath bitwidth. Our future work includes development of efficient algorithms to find the optimal datapath bitwidth.

References

[1] http://kasuga.csce.kyushu-u.ac.jp/˜codesign/

[2] http://www-suif.stanford.edu/

[3] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*, 2nd edition. Morgan Kaufmann Publishers, Inc., 1996.

[4] S. Mahlke, et. al., "Bitwidth Cognizant Architecture Synthesis of Custom Hardware Accelerators," *IEEE Trans. CAD*, vol. 20, no. 11, pp. 1355–1371, Nov. 2001.

[5] B. Shackleford, et. al., "Memory-CPU Size Optimization for Embedded system Designs," *DAC*, 1997.

[6] M. Stephenson, J. Babb, and S. Amarasinghe, "Bitwidth Analysis with Application to Silicon Compilation," *PLDI*, 2000.

[7] H. Yamashita, H. Tomiyama, A. Inoue, F. N. Eko, T. Okuma, and H. Yasuura, "Variable Size Analysis for Datapath Width Optimization", In *Proc. of Asia Pacific Conf. on Hardware Description Languages*, pp. 69–74, 1998.

[8] H. Yasuura, H. Tomiyama, A. Inoue and F. N. Eko, "Embedded System Design Using Soft-core Processor and Valen-C", *IIS J. Info. Sci. Eng.*, vol. 14, pp.587-603, Sept. 1998.

[9] V. Živojnović, J. M. Velarde, C. Schlager, and H. Meyr, "DSPstone: A DSP-oriented benchmarking methodology," *Proc. of Int'l Conf. on Signal Processing and Technology*, 1994.