G-MAC: An Application-Specific MAC/Co-Processor Synthesizer

Alex C.-Y. Chang, Wu-An Kuo, Allen C.-H. Wu and TingTing Hwang
Computer Science Department, Tsing Hua University
Hsin-Chu, Taiwan 30043

Abstract:
A modern special-purpose processor (e.g., for image and graphical applications) usually contains a set of instructions supporting complex multiply-operations. These instructions perform a variety of multiply-operations with various data bit-widths and concurrent-execution requirements. For instance, such an instruction set may include instructions to perform signed/unsigned 32X32, signed/unsigned dual 16X16, signed/unsigned 8X8 MAC, and etc. Typically, a co-processor or a complex MAC (Multiplier-ACcumulator) unit is required to execute those instructions.

Developing such a complex MAC/co-processor involves a series of design tasks including micro-architecture design, component allocation/binding, interconnect binding, pipeline insertion and control generation. This design process is non-trivial, time-consuming and error-prone, which is usually performed by experienced design engineers. In this paper, we present a synthesis method for application-specific MAC/co-processor generation.

The MAC/co-processor synthesis problem is defined as: Given a set of instructions and the number of execution cycles for each instruction, generate a MAC/co-processor design (including a data-path and a control unit) such that the total area-cost is minimized subject to the given execution-cycle constraints.

The MAC/co-processor generation consists of the following two steps. In the first step, we determine a set of minimum-cost components required to realize the given instruction set. In the second step, we perform micro-architectural-level synthesis tasks, including component mapping, interconnect synthesis, pipeline insertion, and control synthesis to generate the MAC/co-processor design.

1. The Minimal-cost Component-set Determination (MCD) Algorithm

We first present several properties that will be used as the foundation of the MCD algorithm.

Observation 1: A type-*n* component with a bit-width of *b* can realize (cover) any instruction of type-*n* operation with a bit-width less than or equal to *b*.

Lemma 1. Let an M-bit multiplicand be decomposed into i sub-operands of bit-width $m_1, m_2, \ldots m_i$ and an N-bit multiplier be decomposed into k sub-operands of bit-width $n_1, n_2, \ldots n_k$. We need ($i \times k$) multipliers $m_1 \times n_1, m_1 \times n_2, \ldots m_i \times n_k$, and ($i \times k$ -1) adders to implement the $M \times N$ multiply-operation.

For Lemma 1, if we set $i \leq 2$ and the multiplier to k sub-operands of the same bit-width, we will have the following property.

Observation 2: Let an M-bit multiplicand be decomposed into 1 or 2 sub-operands and an N-bit multiplier be decomposed into k sub-operands of equal bit-width. Then, there exist *M* possible design alternatives (i.e., *M* decomposition forms with different combinations of multipliers and adders) to implement the multiply-operation.

This research was supported by grants of NSC 91-2215-E-007-040 and NSC90-2218-E-007-043.

Observation 3: Using a single $M \times N$ multiplier to implement a single-cycle (e*k=1*) $M \times N$ multiply-operation is the cheapest implementation in terms of the area cost, i.e., the total gate count.

We formulate the MCD problem into a covering problem. Alg. 1 shows the MCD algorithm. The inputs to the algorithm include a set of instructions (*Inst_Set*) and the number of execution-cycles for each instruction. The output is a minimal-cost component-set (*MC*) that can realize the given instruction set. We will use the example shown in Figure 1 as a walkthrough example, which includes four instructions: *dualmult(16X16,1), dualmult(18X14,1), mult(24X24,1) and mult(32X32,2),* to explain the MCD algorithm.

First, the algorithm partitions the instructions into three groups (Line3): dual (I_{dual}), single-cycle (I_{single}) and multi-cycle (I_{multi}) instructions, follows by sorting the instructions in an ascending order according to their bit-widths (Line 4). The algorithm will perform the covering procedure on the instructions by this order.

Next, the algorithm applies observations 1&3 on the dual-instruction group (Line 5). Note that intuitively more components will be allocated for dual-instructions, which will provide more resource-sharing opportunities for the implementation of single- and multi-cycle instructions. For example, initially it allocates two 16X16 multipliers to realize the *dualmult(16X16,1)* instruction, as illustrated in Figure 1(a). Now consider the *dualmult(18X14,1)* instruction, by applying observation s 1&3 it allocates two 18X16 multipliers that can cover the two dual instructions.

```
1. Algorithm MCD(Inst_Set)
2. begin
3. {$I_{dual}$, $I_{single}$, $I_{multi}$}=PAR (Inst_Set).
4. Sort_BW($I_{dual}$, $I_{single}$, $I_{multi}$);
5. MC=Obser1&3 ($I_{dual}$);
6. for $\forall$ $inst_i \in I_{single}$ do
7. begin
8.   MF = Obser2($I_{single}$);--(k is set to 1)
9.   $MC_{current}$ = Covering(MC, $mf_1$) ;
10.  for $\forall$ $mf_j \in MF$ do
11.    $MC_{temp}$ = Covering(MC, $mf_j$ );
12.    If area_cost($MC_{current}$) > area_cost($MC_{temp}$)
13.    $MC_{current}$ = $MC_{temp}$;
14.  end_for
15.  MC= $MC_{current}$ ;
16. end_for
17. for $\forall$ $inst_i \in I_{multi}$ do
18. begin
19.   MF = Obser2($I_{multi}$);--(k is set to ek)
20.   $MC_{current}$ = Covering(MC, $mf_1$ ) ;
21.   for $\forall$ $mf_j \in MF$ do
22.     $MC_{temp}$ = Covering(MC, $mf_j$ ) ;
23.     If area_cost($MC_{current}$) > area_cost($MC_{temp}$)
24.     $MC_{current}$ = $MC_{temp}$;
25.   end_for
26.   MC= $MC_{current}$ ;
27. end_for
28. return MC;
30.end
```

Alg. 1: The MCD algorithm.

Then, the algorithm first invokes the decomposition-form generation procedure to generate all decomposition-forms for one single-cycle instruction (Lines 6-16). It then applies the covering procedure on all decomposition-forms of each instruction. The covering solution with the minimum area cost will be selected as the final covering result. This covering procedure repeats for all single-cycle instructions. For example, now consider the single-cycle instruction *mult(24X24,1)*. By applying observation 2, we can obtain 24 decomposition-forms. Figure 1(b) shows two covering solutions on two out of 24 decomposition-forms of *mult(24X24,1)*. The first one (on the left) uses two 24X12 multiply-operations to implement the instruction, and the final covering result requires two multipliers and one adder. The second one (on the right) uses a 24X24 multiply-operation to implement the instruction, which requires two multipliers. By computing the overall area costs for all possible covering solutions, the last one has the lowest area cost that will be selected as the covering result.

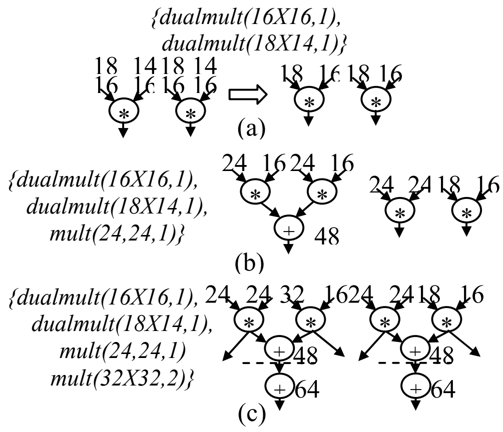

Figure 1: An MCD example.

Next, the algorithm invokes the decomposition-form generation procedure to generate all the decomposition-forms for one multi-cycle instruction (Lines 17-27). Note that k is set to the number of execution cycles of the instruction; i.e., *ek=k*. It then applies the covering procedure on all decomposition-forms of each instruction. This covering procedure repeats for all multi-cycle instructions. For the multi-cycle instructions, the covering procedure needs to take into the cycle constraint (e*k*) into consideration for resource sharing. Consider the two-cycle instruction *mult(32X32,2)*. By applying property 2, we can obtain 32 decomposition-forms. We can use two 32X16 multiply-operations to implement this instruction. Since the cycle constraint e*k=2*, we need only one 32X16 multiplier to realize it. Hence, the covering result is shown in Figure 1(c) (on the left). We can also use four 16X16 multiply-operations to implement the same instruction, which can be realized by using two 16X16 multipliers. The covering result is shown in Figure 1(c) (on the right). The algorithm computes the overall area costs for all possible covering solutions, the one with the lowest area cost will be selected as the covering result. Finally, the algorithm returns the component-set (*MC*).

2. Experimental Results

We have implemented the MCD algorithm and the MAC/co-processor synthesizer *G-MAC*. The inputs to the generator include an instruction set, cycle constraints and the pipeline stages. The outputs include a Verilog RTL description and its synthesis script file. For all experiments, we used Synopsys's *Design Compiler* to synthesize the RTL design into a gate-level design with the maximum-speed option. Then, we used AVANTI's *Apollo* to perform the place & route design tasks. In the experiments, we used the TSMC 0.35um library.

Table 1

#of pipeline stage	(0C, 3.6V)		(25C, 3.3V)		(100C, 2.7V)	
	Timing (ns)	Area	Timing (ns)	Area	Timing (ns)	Area
1	5.98	16401	7.22	16559	10.28	16583
2	4.10	12159	5.26	12967	9.52	13207
3	3.42	13247	5	13504	6.62	14546
Instructions: *mult(32,32,2), mult(32,16,1), dualmult(16,16,1)*						

Table 2

# of pipeline stage	(0C, 3.6V)		(25C, 3.3V)		(100C, 2.7V)	
	Timing (ns)	Area	Timing (ns)	Area	Timing (ns)	Area
1	6.15	19026	8.16	20214	12.52	21847
2	5.06	17115	6.24	18851	9.64	19014
3	4.72	17606	5.78	17976	7.02	18658
Instructions: *mult(32,32,2), mult(28,28,2), mac(24,24,2), mult(32,16,1), mac(16,16,1), mac(12,12,1), dualmac(14,8,1), dualmac(8,8,1), dualmult(16,16,1), dualmult(8,8,1)*						

We have generated two MAC units to realize two instruction sets. The first instruction set includes three instructions and the second set includes 10 instructions, as depicted in Tables 1 and 2. The run-times for the MAC generation were less than one second on a Sun Blade 1000 workstation. Tables 1 and 2 show the experimental results. The results show that under the normal condition (25C, 3.3V) the two designs achieved speeds of 170-200MHz with three pipelined-stages. Figure 2 illustrates the final layout of the first MAC design.

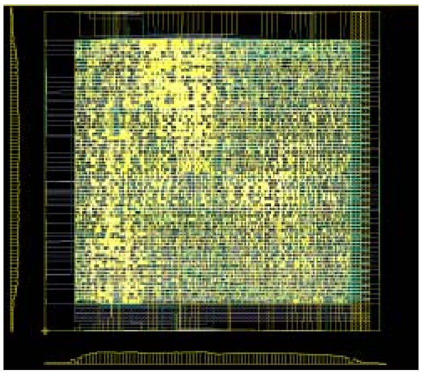
Figure 2: The final layout of the first MAC.

3. Conclusions

In this paper, we have presented a novel technique to determine a minimal-cost component-set for realizing a set of complex multiply-operations. We have also presented a synthesis system *G-MAC* that can automatically generate a complex MAC/co-processor from a given set of instructions and constraints. The experimental results have demonstrated that our proposed method and system can produce high-performance complex MAC/co-processors on the fly.