Code Generation for a DSP Processor*

Wei-Kai Cheng Youn-Long Lin
Department of Computer Science
Tsing Hua University
Hsin-Chu, Taiwan 30043, R.O.C.

Abstract
We propose a method for compiling an application program into microcodes of a programmable DSP processor. Since most state-of-the-art DSP processors feature some sort of parallel processing architectures, the code generation is a non-trivial task. Based on several scheduling and allocation techniques previously developed by the CAD community for high-level synthesis, we propose a DSP code generator. We emphasize on reducing the memory access and register usage conflicts, which often lengthen the total execution time. Starting with an as soon as possible scheduling without regarding to the resource constraints, we transform the scheduling step-by-step into a legal one. In the mean time, registers are allocated and re-allocated for variables taking into account both the memory access and register usage constraints. A software system called THEDA.DSPCG has been implemented and tested using a set of benchmark programs. Simulation of the generated codes targeted towards the TI TMS320C40 DSP processor shows that the proposed approach is indeed very effective.

1 Introduction
Many applications require performing various digital signal processing (DSP) algorithms, which are very computational intensive. When a pure software implementation (on a general-purpose processor) cannot deliver the desired level of performance, designers have to seek solutions in either ASICs or programmable DSP processors. While the ASIC solution can achieve the highest level of performance, the programmable DSP processor approach can lower the design cost and shorten the design time. In both cases, the microcode generation problem is very important as it significantly affects the overall performance.

In either a DSP-oriented ASIC or a programmable DSP processor, multiple, pipelined functional units, as well as multiple-port memory and register files are commonly used. Therefore, a microcode generator must try to exploit the available parallel processing capacity.

In Paulin's survey [3] regarding future CAD requirements by a group of designers, code generation and microcode synthesis are among the most urgent needs. Paulin's survey reports that at least one third of the DSP applications are implemented with programmable DSP processors. This further stresses the importance of the code generation problem. In the past decade, we have witnessed a large amount of research effort being spent on the scheduling and allocation techniques [4] in general and DSP-oriented ASIC synthesis in particular [5]. However, only very little attention has been paid to the microcode generation for programmable DSP processors. Since both the DSP applications and the processor architectures are much different from those of a general-purpose processor, we cannot rely on the advance of the general compiler technology alone to help us fully utilize the computation power of the DSP processors. Instead, we should develop techniques specifically targeted towards the programmable DSP processor architecture.

Code generation is also important in another emerging field of CAD, namely, hardware/software co-design [2]. An effective code generator allows for more functionality to be performed in the software side and, hence, reduce the overall hardware cost.

Instruction scheduling and register allocation are two of the most important subtasks in a code generation procedure. Instruction scheduling rearrange the instructions such that the hardware resources are better utilized, and, hence, the run-time reduced. Register allocation assigns operands to registers. In high-level synthesis, scheduling [6] [8] and register allocation [9] considers only the function unit and bus constraints. But for code generation, there are many other constraints, which further complicates the problem.

In this paper, we integrate several scheduling and allocation techniques previously developed by the high-level synthesis community into a code generator targeted towards a (commercially available) programmable DSP processor. In addition to minimizing the number of control steps, we aim at reducing conflicts due to simultaneous accesses to registers or memory by multiple instructions executed in the pipelined mode. For register allocation, a bipartite weighted matching approach similar to that of Reference [9] is applied iteratively. While the work in Reference [9] aims at minimizing the interconnection complexity, our goal is to maximize the possibility of scheduling more instructions into a control step under the given constraints.

Our algorithm composes of two phases. In the first phase, instructions are scheduled as soon as possi-
nable without regarding to the resource constraints. In the second phase, instructions which violate the constraints are rescheduled and variables are assigned to registers.

The rest of this paper is organized as follows. Section 2 describes the target DSP processor architecture. Section 3 formulates our problem and lay out some assumptions. The proposed code generator, called \textit{THEDA.DSPCG}, is described in section 4. In section 5, some experiment results are presented. Finally, in section 6 we draw a conclusion and point to some directions for possible future research.

2 Target Architecture

Compared with general-purpose microprocessors, DSP processors usually implement special architectural features to support primitive operations commonly found in DSP applications. For example, structured pipelining, MAC (Multiplication-and-Accumulation) unit, and multiple memory ports are all useful and commonly found features in a DSP processor. These architectural uniqueness lead to requirements on special consideration during code generation. In this section, we introduce the architecture and the microcode style of the DSP processor towards which our code generator is targeted.

The first architectural feature is structured pipelining. A DSP processor usually has four stages in the pipeline: instruction fetch, instruction decode, operands fetch, and execute. Operands from the memory are fetched during the operands fetch stage, while those from the register files are fetched during the execute stage.

The second architectural feature is in supporting loop execution over large array of data. For example, in the Fourier transform or image processing, a large array is needed and there cannot be enough registers to hold these data. So most DSP processors have on-chip memory.

In some DSP processors (e.g., TI TMS320C3X, TMS320C4X), multiple instructions can be packed into a single microcode word for simultaneous execution. Unlike in a Very Long Instruction Word (VLIW) architecture, where every instruction packed into a microcode word has full access to all registers, in a DSP processor, the access range may be limited. Consider the microcode format depicted in Figure 1 for the TI TMS320C4X processor. A microcode can contain either a single instruction (Figure 1(a)) or two parallel instructions (Figure 1(b)). The operation field alone determines the number of instructions to be performed by executing the microcode. For the two-instruction case, the number of bits for specifying an operand is halved. Therefore, the addressable ranges are smaller. Furthermore, when a non-load/store instruction is allowed to access the memory (i.e., a memory-to-memory architecture) the usable addressing modes are limited in the two-instruction format. These together make both the scheduling and allocation problems more difficult.

<table>
<thead>
<tr>
<th>operation</th>
<th>dst</th>
<th>src1</th>
<th>src2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(a)

<table>
<thead>
<tr>
<th>operation</th>
<th>dst1</th>
<th>dst2</th>
<th>src1.1</th>
<th>src2.1</th>
<th>src1.2</th>
<th>src2.2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

(b)

Figure 1: Microcode format of a DSP processor: (a) One-instruction microcode, (b) Two-instruction microcode.

3 Problem Formulation

Our problem is to pack instructions into microcode word and allocate registers for variables such that the execution delay caused by either register or memory access conflicts is minimized.

We assume that there are four stages in the structure pipeline: instruction fetch, instruction decode, operand fetch from memory, and operand fetch from register and execute. We further assume that there can be two data accesses from the memory within an instruction cycle. \text{Rx} (x = 1, 2, \ldots) is a data register, while \text{ARx} (x = 1, 2, \ldots) an addressing register.

Figure 2 illustrates a case of register conflict. The second instruction (MPYF) must fetch from the memory an operand whose address is stored in AR2. In the fourth step, the second instruction cannot read its operand from the memory because the address has not been loaded into AR2 yet. Therefore, it must wait for two cycles till its previous instruction has completed.

<table>
<thead>
<tr>
<th>PIPELINE OPERATION</th>
</tr>
</thead>
<tbody>
<tr>
<td>next</td>
</tr>
<tr>
<td>n+1</td>
</tr>
<tr>
<td>n+2</td>
</tr>
<tr>
<td>n+2</td>
</tr>
<tr>
<td>n+3</td>
</tr>
<tr>
<td>n+3</td>
</tr>
</tbody>
</table>

Figure 2: An example of register conflict.

Figure 3 illustrates a case of memory conflict. The simultaneous execution of two STF instructions results in two data transfers from registers to the memory in the fourth control step. Hence, the third instruction (ADDF) cannot fetch its operand from the memory in the fourth cycle, resulting in a one-cycle delay. There are many other cases leading to memory conflicts. Due to the paper length limitation, we will not describe them all here.

The number of conflicts can be reduced with careful instruction scheduling and register allocation. For example, in Figure 3, if we pack into one microcode
word the instruction ADDF together with any one of the two STF instructions followed by the other STF instruction, there will be no delay in execution.

In all, our problem is to schedule instructions and allocate registers subject to the following constraints:

- **Constraints on the number of function units**
  This constraint determines the number of instructions that can be issued simultaneously.

- **Constraints on the number of registers**
  This constraint defines the number of data registers, addressing registers, and other special purpose registers. In addition, the data types that can be stored in each class of registers are also defined.

- **Constraints on the number of I/O ports of the register files**
  For example, if two instructions are issued as in Figure 1(b) and there are two read ports, then at most two of the four source operands can be in the direct register mode, and the other two must come from the memory. Any two instructions together having more than two register source operands cannot be issued simultaneously.

- **Constraints on the number of memory banks and buses**
  This is similar to the previous constraint, i.e., the number of memory banks and buses together set a limit on the number of simultaneous memory accesses.

- **Constraints on register usage**
  Some processors may have more restricted usage of registers. For example, the microcode style depicted in Figure 1(b) allows only a subset of registers for each operand when multiple instructions are executed simultaneously.

## 4 Scheduling and Allocation

Instruction scheduling and register allocation are two of the most important subtasks in DSP code generation. Traditionally, they are performed separately because of the otherwise high problem complexity. In order to achieve better results, a more global approach that considers the two subtasks simultaneously is needed. In the following subsections, we will first propose such an algorithm and then describe our considerations when defining cost functions for the algorithm.

### 4.1 Algorithm Overview

```
Algorithm rescheduling \( (S_{DG}, II, C_s) \)
while violation-flag is true do
  violation-flag <- false;
  \( S_{index} = 1 \);
  while \( S_{index} \leq II \) do
    if there exists violation in state \( S_{index} \) then
      violation-flag <- true;
      \( U = \) the set of instructions involved with violations;
      let \( Op \in U \) be the instruction with the highest priority;
      reconfig(\( Op \));
      break; /* Break the inner while loop */
    else
      \( VI = \) the set of input variables in state \( S_{index} \);
      \( VO = \) the set of output variables in state \( S_{index} \);
      register allocation(\( VI, VO \);
      if there is violation against the register usage constraint then
        violation-flag <- true;
        \( R = \) the set of instructions whose variables involve with register violation;
        let \( Op \in R \) be the instruction with the highest priority;
        reconfig(\( Op \));
        break; /* Break the inner while loop */
      else
        \( S_{index} ++ \);
        endif
      endwhile
    endif
  endwhile
  return(\( S_{DG} \));
end rescheduling.
```

Figure 4: The rescheduling algorithm.

Our algorithm is a transformation-based heuristic. It schedules instructions into control steps and assigns variables to registers during the transformation process. The algorithm consists of two phases. In the first phase, instructions are scheduled as soon as possible. Only the data dependency constraint is taken into account while all other types of constraints are totally ignored. What obtained after this phase is a maximally parallel schedule. During the second phase, a rescheduling procedure is applied to resolve all constraint violations. In the mean time, register allocation (and re-allocation) is performed using a bipartite weighted matching algorithm.

The first phase is trivial and, thus, will not be described here. Figure 4 shows the algorithm of the second phase. Inputs to the algorithm include the as soon as possible instruction scheduling \( S_{DG} \) from the first phase, the initiation interval \( II \) (of the loop), and the set of constraints \( C_s \). The output is a violation-free instruction scheduling with register allocation information. For every state, we first check whether there
exist violations against the resource constraint. If it is the case, an instruction is rescheduled to the next state. The selection of instruction for rescheduling is based on a priority function to be defined in the next subsection. If there exists no resource constraint violation, a register allocation algorithm is used to (1) assign output variables to registers, (2) check input variable assignment, and (3) re-allocate registers to meet the register usage constraints. If a feasible solution emerges, scheduling of the current state is done and the algorithm repeats itself for the next state. Otherwise, the algorithm selects another instruction for rescheduling and performs register allocation and re-allocation for all previously encountered states. The algorithm terminates itself when no more states violating the constraints.

Algorithm register_allocation \((VI, VO)\)

\[
\text{reg.vio} \leftarrow \text{false}; \\
\text{matching}(VO); \\
\text{if there is violation against register usage constraint then} \\
\quad \text{reg.vio} \leftarrow \text{true}; \\
\text{else} \\
\quad \text{if some variables in } VI \text{ violate the register usage constraints then} \\
\quad \quad \text{re-allocating}(VI); \\
\quad \quad \text{if some variables in } VI \text{ violate the register usage constraints then} \\
\quad \quad \quad \text{reg.vio} \leftarrow \text{true}; \\
\quad \text{endif} \\
\text{endif} \\
\text{end} \text{register.allocation.}
\]

Figure 5: The register allocation algorithm.

Figure 5 shows the register allocation algorithm. First, it assigns the output variables of the current state to registers using a bipartite weighted matching algorithm [7]. If no legal assignment can be found, the procedure terminates itself (i.e., a failure). On the other hand if a legal assignment has been found, the algorithm proceeds to check whether the previously assigned input variables of the current state have violated the register usage constraint. If the assignment is illegal, we try to eliminate the violations by re-allocating these variables to some other registers. The algorithm reports failure when no re-allocation can eliminate all violations.

Figure 6 shows the register re-allocation algorithm. First, it finds the set of states in which the variables in \( VI \) are produced. Then, it re-allocates the output variables in those states using the bipartite weighted matching algorithm one state at a time. If this re-allocation cannot completely eliminate the violations caused by input variable assignment, they can only be resolved by instruction moving as described in Figure 4.

Algorithm re-allocation \((VI)\)

Let \( S \) = the set of states having at least one output variable \( var_o \in VI \) for every state \( s_i \in S \) starting with the earliest state do

\[
VO' = \text{the set of output variables in state } s_i; \\
\text{matching}(VO'); \\
\text{endfor} \\
\text{end re-allocation.}
\]

Figure 6: The re-allocation algorithm.

4.2 Instruction Rescheduling Considerations

There are several criterion in defining the cost function for selecting an instruction for rescheduling:

- **chances of reducing violations**
  1. If its rescheduling will increase the total number of control steps, an instruction is not a good candidate and, therefore, has a lower priority.
  2. An instruction with the higher freedom between the current scheduling and its as late as possible scheduling should be given the higher priority for rescheduling.
  3. An instruction whose rescheduling is less likely to cause other violations should be given a higher priority for rescheduling. This is similar to that of Reference [8].

- **register usage**
  If its current scheduling ends the lifetime of a variable, an instruction is not a good candidate for moving. On the other hand, if its current scheduling consume a new register, an instruction should be moved away from the current state.

- **reducing register and memory conflicts**
  If its rescheduling can reduce the conflicts, an instruction should be given a higher priority for rescheduling. On the other hand, if its rescheduling will introduce new conflicts, an instruction should not be moved.

4.3 Register Allocation Considerations

This subsection describes the criterion used by the algorithm during register allocation for output variables. These criterion are expressed as the edge weight of the bipartite graph for matching. They are summarized as follow:

- **legal assignment**
  Every variable must be assigned to a register under the register usage constraint. During register re-allocation, in addition to be legal in the current state, a register assignment must not void other previously legal assignment. Consider the example depicted in Figure 7. After both scheduling
and allocation have been done for states 1 and 2, we detect that an input variable of instruction 6 violates the register usage constraint. Therefore, we have to perform a register re-allocation for the output variables of state 1. During re-allocation, we must legally assign variables to resolve the violations associated with instruction 6. In addition, we must not make the input variables of instruction 5, which have been previously legal, become illegal.

- **violation reduction**
  Because the output variable of an instruction will become the input variable of some other instructions, during register allocation we must also predict whether an assignment will cause register usage violations in the later stage. The allocation that is less likely to cause violations is preferred.

- **register usage**
  In addition to legal assignment and violation reduction, we prefer assigning a variable to a register with more usage constraint. For example, if the registers accessible in Figure 1(a) and (b) are R1–R12 and R1–R8, respectively, then the set of registers R9–R12 have more usage constraint. If we can legally assign a variable to a register with more usage constraint, then we can save more less-constrained registers for later use.

![Figure 7: A DFG for illustrating register re-allocation.](image)

5 Experimental Results

We have implemented the proposed approach in a software system called THEDA.DSPCG using the C programming language on a SUN Sparc-II workstation. We have tested THEDA.DSPCG with a set of benchmark programs. We compare THEDA.DSPCG and the TI TMS320C40 C compiler [12] in terms of the code efficiency targeted towards the TMS320C40 DSP processor.

Five frequently used algorithms in the C language from Reference [10] are used as the benchmarks. They are a 6th order FIR filter (fir), an IIR filter (iir), a 16-point discrete fourier transform (dft) algorithm, a fast fourier transform (fft) algorithm, and a 32-by-32 two-dimensional convolution algorithm with a 3rd order Kaiser filter kernel (convolution). We use the TI TMS320C40 simulator [13] to simulate the codes and count the execution cycles. Table 1 shows the total execution cycles of the codes generated by both THEDA.DSPCG and the TI 320C40 compiler.

<table>
<thead>
<tr>
<th></th>
<th>fir</th>
<th>iir</th>
<th>dft</th>
<th>fft</th>
<th>conv.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TI</td>
<td>18</td>
<td>42</td>
<td>2322</td>
<td>1089</td>
<td>41400</td>
</tr>
<tr>
<td>THEDA.1</td>
<td>15</td>
<td>36</td>
<td>2968</td>
<td>1023</td>
<td>41400</td>
</tr>
<tr>
<td>THEDA.2</td>
<td>15</td>
<td>33</td>
<td>2752</td>
<td>1020</td>
<td>41400</td>
</tr>
<tr>
<td>THEDA.3</td>
<td>15</td>
<td>33</td>
<td>2720</td>
<td>987</td>
<td>41400</td>
</tr>
</tbody>
</table>

Table 1: Comparison with TI’s processor in terms of execution cycles. THEDA.1 : THEDA.DSPCG/nr; THEDA.2 : THEDA.DSPCG/np; THEDA.3 : THEDA.DSPCG/all.

The first row of Table 1 shows the execution cycles of the codes generated by the TI TMS320C40 compiler with the highest level (Level 2) optimization. The second row (THEDA.DSPCG/np) shows the execution cycles of the codes generated by THEDA.DSPCG when the register allocation refinement procedure (Figure 6) is deactivated. The third row shows the case when we did not pay attention to the reduction of conflicts (THEDA.DSPCG/np). When all optimization procedures are activated the results are the best and shown in the fourth row (THEDA.DSPCG/all).

Figure 8 shows the program listings for the iir benchmark. Figure 8(a) is the C source program from Page 174 of Reference [10]. Figure 8(b) is the TMS320C40 assembly code generated by the TI TMS320C40 compiler using Level 2 (the highest level) optimization. Figure 8(c) shows the assembly code generated by THEDA.DSPCG with all optimization procedures activated. Compared Figure 8(b) and (c), we can see that instruction ordering and register allocation are all different. In Figure 8(c) we can pack three two-instruction microcodes, while in Figure 8(b) we have no similar opportunity.

As described in the previous section, except the function unit constraint, there are other constraints for operations that can be specified in the same instruction cycle. In our observation, the function units are not fully utilized in the result of Table 1. This means that there must be other constraints such that the operations can not be more compacted in the same cycle.

6 Conclusion and Future Work

We have presented an approach to the code generation problem for a DSP processor. In addition to minimizing the cycle count, we emphasize on minimizing conflicts caused by simultaneous accesses to registers and the memory. We also take register usage constraints into account during code generation. By tightly integrating instruction scheduling and register allocation (and re-allocation), we are able to deal with the interdependency between these traditionally partitioned subtasks. We have designed cost functions reflecting all practical considerations. Experimental
for (i = 0; i < ir->length; i++) {
    history1 = * hist1_ptr;
    history2 = * hist2_ptr;
    output = output - history1 * (* coef_ptr++);
    new_hist = output + history2 * (* coef_ptr++);
    output = new_hist + history1 * (* coef_ptr++);
    * hist1_ptr++ = * hist2_ptr;
    * hist1_ptr++ = new_hist;
    hist1_ptr++;
    hist2_ptr++;
}

(a)

LDF * AR4, R10
MPYF * AR2++, R10, R0
SUBF R0, R9
LDF * AR5, R2
MPYF * AR2++, R2, R3
SUBF R3, R9, R3
MPYF * AR2++, R10, R9
ADDF R3, R9
MPYF * AR2++, R2, R0
ADDF R0, R9
STF R10, *AR5+
STF R3, * AR4+
ADDI 1, AR4
ADDI 1, AR5

(b)

LDF * AR4, R5
LDF * AR5, R4
MPYF * AR2++, R3, R6
STF R5, * AR5+
SUBF R6, R9, R3
MPYF * AR2++, R4, R0
SUBF R0, R3, R0
MPYF * AR2++, R5, R1
STF R0, * AR4+
ADDF R0, R1, R3
MPYF * AR2++, R4, R1
ADDF R1, R3, R3
ADDI 1, AR4
ADDI 1, AR5

(c)

Figure 8: Program listings for the iir benchmark: (a) The C source code, (b) The assembly code generated by the TI compiler, (c) The assembly code generated by THE DA.DSPCG.

results over a set of real DSP kernel programs have shown that our proposed software system is able to generate very efficient code compared with a state-of-the-art commercial compiler.

During our experiment process, we found out that source-level program transformation may lead to better code efficiency. Therefore, DSP-processor-oriented source transformation is an interesting area for further research.

From now on we would also like to study the code generation problem under a multiple-DSP-processor environment. Such an environment promises even higher level of performance. In addition to the proposed techniques, we will have to develop methods for partitioning and interface/communication design.

References


