Register Assignment through Resource Classification for ASIP Microcode Generation

Clifford Liem, Trevor May, Pierre Paulin

Bell-Northern Research Ltd.
Ottawa, ON, Canada
E-mail: cbl@bnr.ca tmay@bnr.ca paulin@bnr.ca

Abstract

Application Specific Instruction-Set Processors (ASIPs) offer designers the ability for high-speed data and control processing with the added flexibility needed for late design specifications, accomodation of design errors, and product evolution. However, code generation for ASIPs is a complex problem and new techniques are needed for its success. The register assignment task can be a critical phase, since often in ASIPs, the number and functionality of available registers is limited, as the designer has opted for simplicity, speed, and low area. Intelligent use of register files is critical to the program execution time, program memory usage and data memory usage. This paper describes a methodology utilizing register classes as a basis for assignment for a particular style of ASIP architectures. The approach gives preference to special purpose registers which are the scarce resources. This naturally leads to the objectives of high speed and low program memory usage. The approach has been implemented in a system called CodeSyn [1] and used on custom ASIP architectures.

1 Introduction

The Application Specific Instruction-Set Processor (ASIP) has recently emerged as a new hardware design solution taking the best of two worlds: flexibility through programmability from the general purpose processor and high-speed, low power, and low area through dedicated circuitry from the Application Specific Integrated Circuit (ASIC). Much as synthesis has struggled to keep up with the rate of advances in process technology, now code generation is challenged by the new design styles in ASIPs.

Register assignment for ASIPs can be a difficult task due to the small number and restricted functionality of physical register files. Limitations on the availabilty of registers is usually motivated through the designer's will for high speed and low area through the reduction of busses and multiplexers. Also, a simple design reduces the possibility of errors and allows fine tuning to an application area.

In this paper, we present an effective approach for the register assignment task in code synthesis for ASIPs. An overlapping classification of register function has been employed to guide the algorithm. The concept of register classes is not new [2][3]; however, they are used in our approach in a unique manner. The methods are particularly effective with a specific ASIP design style, but we anticipate that many of the methods are applicable to other ASIP and DSP design styles.

This approach works in conjunction with the instruction-set matching and selection methodology outlined in [4], and is part of the code generation system called CodeSyn [1]. CodeSyn is one of the tools in the embedded systems development environment, FlexWare [1], which also includes a retargetable VHDL-based instruction-set simulator, Insulin [5].

The complete code generation system also contains a source-level parser, a graph rewrite module, an instruction-set pattern matcher/selector, and a scheduler, which are used prior to register assignment. The back end contains a compactor, assembler and linker. The process is shown in Figure 1 and the steps are outlined in [1]. The current source language is C.

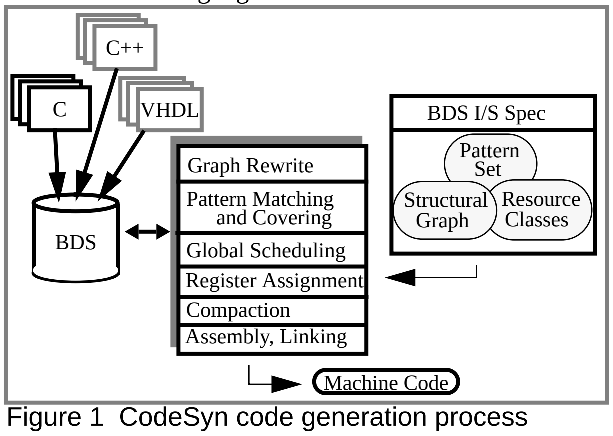


Figure 1 CodeSyn code generation process

This paper is organized as follows. Section 2 outlines related work in this area. Section 3 describes the ASIP design style to which we have targeted this approach and a formulation of the problem. Section 4 describes the register assignment approach. Section 5 describes a full code generation example and results. Section 6 provides a summary and conclusion.

2 Related Work

In high-level synthesis, register allocation has traditionally been defined as the determination of the number of registers needed to store values between time-steps for a given application. The required number of registers are created or drawn from a library. Work in this area has been successfully addressed through techniques such as rule-based systems[6], branch and bound methods [7], linear programming [8], and iterative approaches[9].

In contrast to this, register assignment in code generation involves finding the best use of a *fixed* set of registers among hardware constraints. A popular technique for register assignment is graph coloring [10][11]. The procedure has been formulated as follows. An interference graph is constructed with nodes representing registers and connecting edges representing conflicting assignments. Adjoining nodes are given different colors, then each color is related to a physical register . While this technique can give good results for architectures with regular register files, it does not address dedicated architectures with special purpose registers. In addition, coloring techniques do not naturally address the concept of variable lifetimes in a procedural control context.

Register assignment has also been solved through greedy approaches such as that taken from the routing problem in microchip mask layout [12]. The *left-edge algorithm* is applicable to both hardware synthesis and code generation. It offers optimal assignment for a regular and fixed register file within a control-flow block boundary. We have built upon this algorithm and modified it to use functional classes of registers for one part of our approach for application-specific processors.

As processor architectures become more application specific in nature, new approaches [13][14] are emerging which regard the heavy interdependence between scheduling of instructions and register allocation. For effective code generation, methods such as these will become important precursors to register assignment approaches, such as the one presented here.

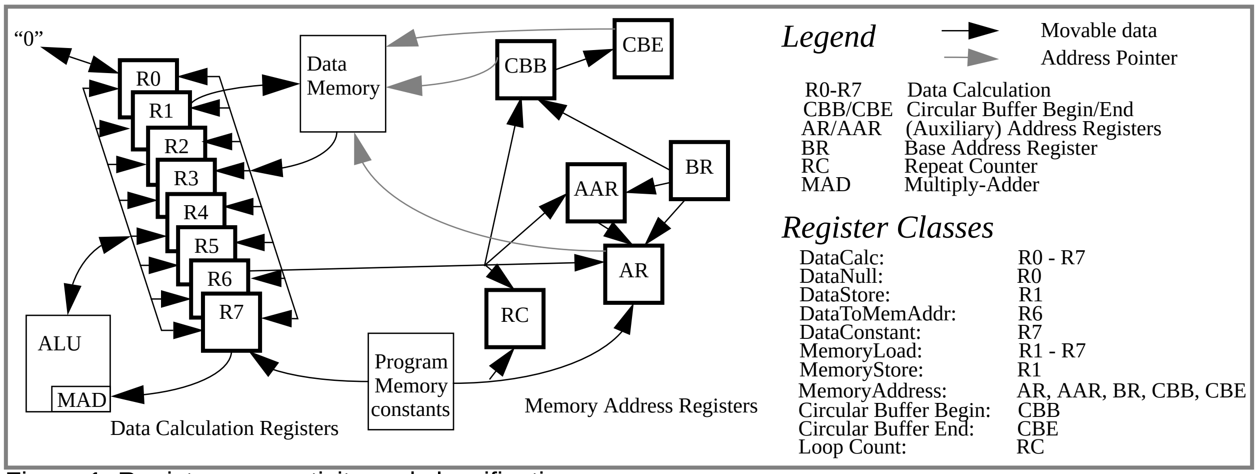
3 Problem Domain

3.1 Target ASIP Architecture

The target architecture-style for this approach has been described in [4] and is repeated here for clarity. The registers and allowable data movement is depicted in Figure 1. Specific registers must be used for particular tasks, motivated by the designer's desire for speed and area improvements. This figure shows that R0 is always a constant 0 source as well as a bottomless sink, R1 is the only register with write access to data memory, R6 is the only data register through which data can be moved from the ALU to the address register, and R7 is the only register which can hold a constant from program memory and serve as a multiplicand in the multiply-adder of the ALU.

3.2 Problem Formulation

We approach register assignment as an isolated problem following selection of instructions which can execute the main operations of the source algorithm, and coarse scheduling of these operations. A coarse schedule gives the general ordering of instructions without regard to the details of parallelism. The task is to determine specific physical registers as the input and output operands of the instructions. Conflicting uses of registers can only be resolved by moves to available registers and spills to memory. In this style of architecture register to register moves are less expensive than spills to memory, as addresses have to be set up for the later. It is rare but possible to have the converse.



R0-R7	Data Calculation
CBB/CBE	Circular Buffer Begin/End
AR/AAR	(Auxiliary) Address Registers
BR	Base Address Register
RC	Repeat Counter
MAD	Multiply-Adder

DataCalc:	R0 - R7
DataNull:	R0
DataStore:	R1
DataToMemAddr:	R6
DataConstant:	R7
MemoryLoad:	R1 - R7
MemoryStore:	R1
MemoryAddress:	AR, AAR, BR, CBB, CBE
Circular Buffer Begin:	CBB
Circular Buffer End:	CBE
Loop Count:	RC

Figure 1 Register connectivity and classification

4 Assignment Approach

4.1 Register Classification

We have adopted a methodology whereby we define a number of overlapping *register classes* for the given architecture. These classes group a set of registers by function. For example, in Figure 1 the register class, *DataCalc*, contains eight physical registers (R0, R1, R2, R3, R4, R5, R6, R7). Data calculation functions can be performed on any of these registers. Similarly, the register class, *DataStore*, contains one physical register (R1), the only register with write access to the data memory. There are typically two levels of register classes: a broad categorization for overall function, and a small categorization for special purpose functions.

As explained in [4], micro-instruction patterns have been annotated with register classes which indicate where data can reside for the input(s) and output(s) of each instruction. This has aided in the selection of micro-instructions as well as providing the allowable register sets for assignment.

4.2 Assignment Process

We begin with one or several general Control/Dataflow Graphs (CDFGs) corresponding to the source application. As shown in Figure 1, the graph has been pattern-matched and covered to instructions of the target, then coarsely scheduled prior to register assignment.

Between operations that transfer data between registers, behavioural ReadRegister and WriteRegister nodes are inserted carrying register classes which have been annotated on the in/out terminals of the patterns. An example is shown in Figure 2. The directed segments on the right of the diagram indicate control-steps, with the dotted lines showing the execution of operations on each step. This data-flow portion of the CDFG has been scheduled for a one ALU processor.

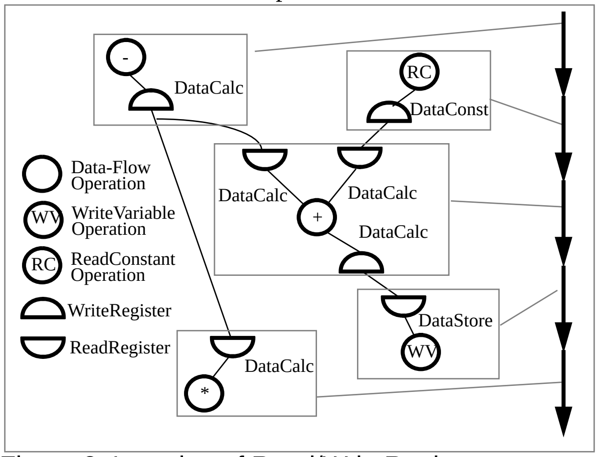


Figure 2 Insertion of Read/WriteRegister Behavioural Nodes

Candidate register sets are calculated for the data-flow between operations from the intersection of the registers in each register class (the register allocation phase of the algorithm). Three examples are shown in Figure 3.

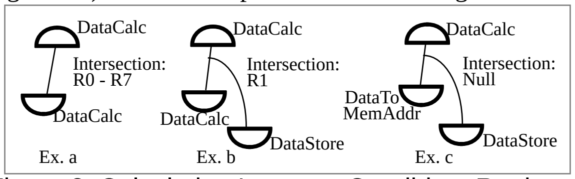


Figure 3 Calculating Intersect Candidate Registers

Note that in Figure 3 Ex. c has an empty candidate intersection which indicates that a register move is needed. Necessary register moves are determined after pattern matching and covering and inserted before scheduling. The insertion point for the move is determined heuristically by a combination of the size of the register classes associated with the ReadRegister nodes and the lifetime of the variable.

After a candidate register list is made for each data-flow connection between operations, assignment begins. The approach is greedy in nature and based upon the left-edge algorithm [12] with improvements. The general procedure is as follows:

1. Assignment begins at intersection points which have register classes containing only one member register.
2. Assignment for the intersection points in Step 1. are ordered based upon lifetime and number of reads. Assignment begins with the shortest lifetime and fewest number of reads.
3. Steps 1 and 2 can lead to register assignment conflicts which are handled by greedily inserting register moves.
4. The remaining candidate intersections are assigned by a left-edge algorithm, inserting spills to memory when there are no available registers to be assigned.

This approach is geared towards giving priority to registers dedicated to specific tasks.

4.3 Assignment of classes with a single member

ReadRegister nodes having a register class containing only one member register (termed single ReadRegisters) are assigned first. Figure 4 shows an example in which the *DataStore* and the *DataToMemAddr* classes contain only one register each (See Figure 1).

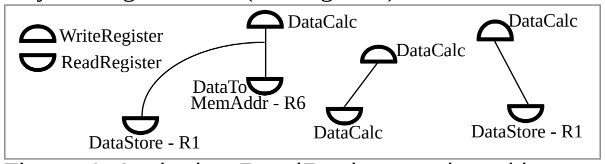


Figure 4 Assigning ReadRegister nodes with single member classes.

Having assigned the single ReadRegisters, an attempt is made to assign all those WriteRegisters which fanout to (or feed) these nodes. Difficulties can arise through two sources:

1. The physical register is in use (has been previously assigned).

If the physical register is in use, the algorithm finds the WriteRegister(a) which is currently using the phyisical register and determines if it occurs before the subject WriteRegister(b). If so, a move is inserted after WriteRegister(a). A move consists of a ReadRegister feeding a DataMove operation feeding a WriteRegister inserted directly after the original WriteRegister. Note that the DataMove operation must later be matched and covered with an instruction-set pattern.

2. A WriteRegister may fanout to more than one ReadRegister, where each ReadRegister may or may not be assigned to a different register.

In this case, the nearest ReadRegisters to the WriteRegister are assigned first, then the rest are resolved in a greedy manner by inserting register to register moves. Heuristically, moves like these *save* the scarce resource of register classes with just a single member.

Following this procedure and in the same fashion, single WriteRegisters are assigned indiscriminately. Unassigned ReadRegisters which fanout from (feed off) any assigned WriteRegisters are attempted to be assigned. Again difficulties arise in the same manner as previously. These are resolved heuristically regarding lifetimes and regarding single member register classes with priority.

It is possible to reverse the two tasks described above by assigning single WriteRegisters before single ReadRegisters. Results are dependent on the number and functionality of the special purpose registers in the architecture. It is also dependent on the applications for which code is being generated.

4.4 Assignment of the remaining Read/Write Registers.

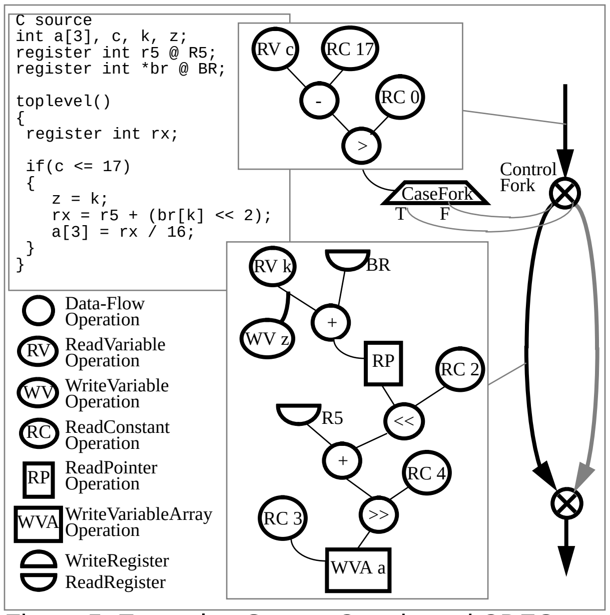
In our approach, the greedy left-edge algorithm is formulated as follows. Starting at the top of the program flow and traversing towards the bottom, each WriteRegister which is visited is assigned to the first available register and marked in use until all fanouts to ReadRegisters are complete. Pre-calculated intersect sets makes this procedure easy.

At points where there are no available registers, spills to memory are inserted. This is nontrivial since the DataStore class is needed and it may be a dedicated register (R1) in some architectures (Section 3.1). We have resolved this problem by placing priority assignment orders on the registers in each class. In this case we have ensured that the DataStore members are regarded as low priority for assignment in other classes. This increases the likelihood of the ability to spill values to memory. As the number of dedicated classes increase, measures such as these become less effective.

For a variable with multiple reads, an inserted spill consists of a store to memory followed by a separate load for each read. It is possible to reduce the number of loads depending on the number of live variables at this critical point. This is an area for future enhancement.

5 Results

We were unable to find a compiler which could successfully and consistently generate correct code for the style of ASIP described in Section 3.1 (that is what motivated this work!). Therefore, having no scale for benchmarks and since this paper describes just one part of code generatioin we have opted to show results through a detailed example. The following is a full example of the CodeSyn code generation process with emphasis on the register assignment procedure. The source C code and corresponding CDFG are shown in Figure 5. Some extensions to ANSI C have been adopted for special features. The *register* storage class has been used heavily with the @ symbol to indicate a specific physical register. Notice that two graph rewrites have been done to the CDFG. The *if* condition has been rewritten to a *subtract followed by a compare to zero*, which allows matching to a *branch on negative* instruction; and, the *divide by 16* has been rewritten to a *right shift by four*. These are rule-based rewrites based on the specific capabilities of the architecture. Also, register *rx* is regarded as a temporary value and replaced by pure dataflow in the CDFG.



```
C source
int a[3], c, k, z;
register int r5 @ R5;
register int *br @ BR;

toplevel()
{
 register int rx;

 if(c <= 17)
 {
   z = k;
   rx = r5 + (br[k] << 2);
   a[3] = rx / 16;
 }
}
```

Figure 5 Example - Source C code and CDFG

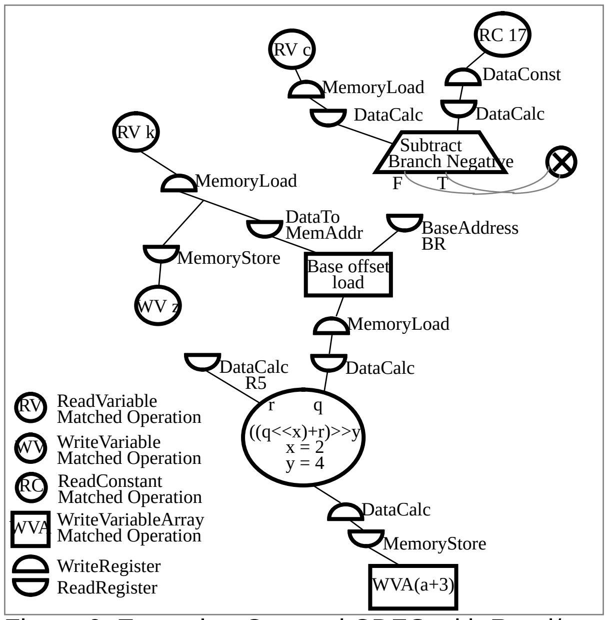


Figure 6 Example - Covered CDFG with Read/WriteRegister Nodes

Figure 6 shows the example after pattern matching and covering to the architecture instruction set. Notice that the pattern matcher has found instructions which maximize the number of operations executed together. Between the operations, behavioral Read/WriteRegister nodes have been inserted. These nodes have been annotated with register classes corresponding to those found on the input/ouput terminals of the matched patterns. Notice that two ReadRegister nodes (BR and R5) have previously been assigned by the user (Figure 5). At this point, these physical registers are checked for inclusion in the annotated register class.

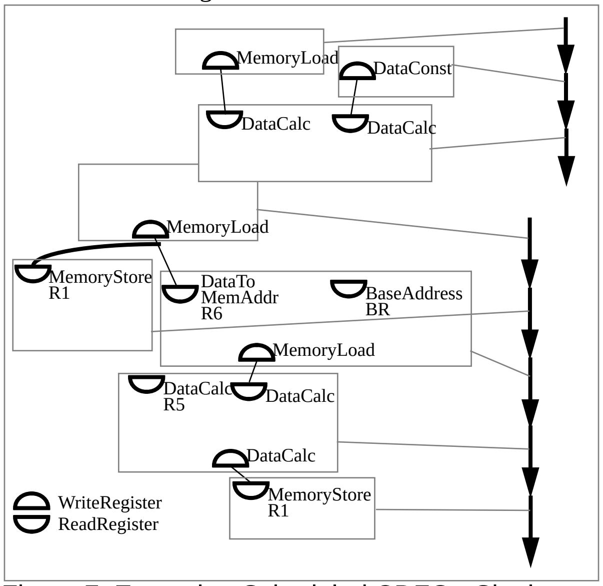


Figure 7 Example - Scheduled CDFG - Single member reads assigned

Figure 7 shows the same CDFG focussing on the Read/WriteRegister operations. The CDFG has been coarsely scheduled based on list methods. Single ReadRegister nodes (register class with one member) are shown assigned to physical registers (i.e. MemoryStore - R1 ; DataToMemAddr - R6), as well as those which have been assigned by the user.

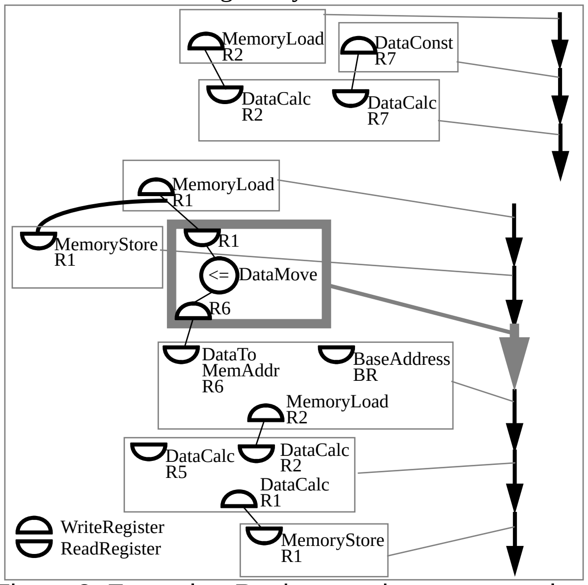


Figure 8 Example - Register assignment complete

Figure 8 shows the CDFG at completion of register assignment. Notice that a conflict move has been inserted to resolve the two ReadRegisters which have been assigned to special purpose registers (R1 and R6), and are written by a common WriteRegister. At this point, the CDFG is completely mapped to the instruction-set of the architecture. Assembly code instructions which have been associated with the patterns are emitted along with the assigned registers. This is shown in Figure 9.

```
 .segment   _toplevel_code
 .global    _toplevel
_toplevel:
 immd_to_ar _c
 ld         R2
 immd_to_r7 17,R7
 sub        R2,R7,R0,0,0
 immd_to_ar _toplevel_L1
 bneg
 immd_to_ar _k
 ld         R1
 immd_to_ar _z
 st         R1
 add        R1,R0,R6,0,0
 r6_plus_br_to_arR6
 ld         R2
 add        R5,R2,R1,2,4
 immd_to_ar _a+3
 st         R1
_toplevel_L1:
 ret
 .segment   reg_example_code_data,memtype=1,wordsize=16
 .global _a
_a:
 .bss       3
 .global _c
_c:
 .zero      1
 .global _k
_k:
 .zero      1
 .global _z
_z:
```

Figure 9 Example - Sequential assembly code

This code can run directly on the machine; however, since the processor allows parallelism, it is possible to compact the code based on the resources used by each micro-operation. The compacted code is shown in Figure 10.

```
/*0x0000*/ CONT ; NOP ;                NOP ;  NOP ;           I_TO_AR #0x003
/*0x0001*/ CONT ; NOP ;                LD R2; MV #0x011,R7;   NOP
/*0x0002*/ CONT ; SUB R2,R7,#0,R0,#0 ; NOP ;  NOP ;           I_TO_AR #0x00B
/*0x0003*/ BNEG ; NOP ;                ST R5; NOP ;           NOP
/*0x0004*/ CONT ; NOP ;                NOP ;  NOP ;           I_TO_AR #0x004
/*0x0005*/ CONT ; NOP ;                LD R1; NOP ;            I_TO_AR #0x005
/*0x0006*/ CONT ; ADD R1,R0,#0,R6,#0 ; ST R1; NOP ;           NOP
/*0x0007*/ CONT ; NOP ;                NOP ;  NOP ;           BR_PLS_R6
/*0x0008*/ CONT ; NOP ;                LD R2; NOP ;           I_TO_AR #0x003
/*0x0009*/ CONT ; ADD R5,R2,#2,R1,#4 ; NOP ;  NOP ;           NOP
/*0x000a*/ CONT ; NOP ;                ST R1; NOP ;           NOP
/*0x000b*/ RET ;  NOP ;                NOP ;  NOP ;           NOP
 SUSP ;           SUSP ;               SUSP ; SUSP ;          SUSP
END.
```

Figure 10 Example - Compacted assembly code

For this example, we were hard-pressed to write more efficient hand-code than CodeSyn had produced.

We have run a number of representative benchmarks through the CodeSyn code generation system. For architectures of the design style described earlier, we were able to generate code within 20% of hand-coded quality on average [4].

6 Conclusion

This paper has introduced a register assignment approach for code generation targeting a style of ASIP architectures with heavy dependence on special purpose registers. The methodology has been proven effective in the code generation system, CodeSyn. Early attempts to retarget a commercial code generation system to the style of architecture described in Section 3 were only partially successful. The solution was to disallow the compiler to use any registers needed for a dedicated function. This made the general data calculation register set severely limited, and so, in many cases code generation was unsuccessful due to lack of registers. Thus, the greatest contribution of this work has been to provide the capability of code generation for this ASIP design-style.

Directions for future work include instruction scheduling with regards to register class usage, and improvements to conflict resolutions. As the organization of registers for ASIP and DSP architectures varies greatly in style, we will be striving for register assignment methods which will be general enough to encompass all varieties. In a recently studied target for CodeSyn, the instruction set shows register class restrictions based upon the limited instruction word length. These restrictions are difficult to manage, and correspond theoretically to *dynamic* register classes. This poses a huge challenge on current register assignment approaches.

Other promising avenues may be in the area of memory management techniques to make best use of available resources before detailed register assignment. Some work in this area has already been investigated [15][16].

7 References

[1] P. Paulin, C. Liem, T. May, S. Sutarwala, "DSP Design Tool Requirements for Embedded Systems: A Telecommunications Industrial Perspective", to appear in *Journal of VLSI Signal Processing* (special isssue on synthesis for real-time DSP), Kluwer Academic Publishers, 1994.

[2] H. Feuerhahn, "Data-Flow Driven Resource Allocation in a Retargetable Microde Compiler", *21st Int. Symposium on Microarchitecture*, 1988, pp. 105-107.

[3] R. Stallman, "Using and Porting GNU CC, version 2.4", distributed with GNU gcc, Free Software Foundation, June 1993

[4] C. Liem, T. May, P. Paulin, "Instruction-Set Matching and Selection for DSP and ASIP Code Generation", *European Design & Test Conf.* Feb 1994, pp. 31-37.

[5] S. Sutarwala, P. Paulin, Y. Kumar, "Insulin: An Instruction Set Simulation Environment", *Proc. of CHDL-93,* April 1993, pp. 355-362.

[6] J. Rabaey, H. DeMan, J. Vanhoof, G. Goossens, and F. Catthoor, "CATHEDRAL-II: a sysnthesis system for multiprocessor DSP systems", in D. Gajski (ed.), Silicon Compilation, Addison-Wesley, 1988, pp. 311-360.

[7] P. Marwedel, "A new synthesis algorithm for the MIMOLA software system", *23rd Design Automation Conf.*, June 1986, pp. 271-277.

[8] C. Gebotys, M. Elmasry, "Simultaneous Scheduling and Allocation for Cost Constrained Optimal Architectural Synthesis", *28th Design Automation Conf.,* June 1991, San Francisco, pp. 2-7.

[9] K. Kucukcakar, A. Parker, "Data path design tradeoffs using MABAL", *the International Workshop on High-level Synthesis*, Kennebunkport, ME, October 1989.

[10] G.J. Chaitin, "Register Allocation and Spilling via Graph Coloring", *ACM Sigplan Notices*, 17, pp. 98-105.

[11] L.J. Hendren, G.R. Gao, E.R. Altman, C. Mukerji, "A Register Allocation Framework Based on Hierarchical Cyclic Interval Graphs", *Int. Conf. on Compiler Construction*, 1992.

[12] F.J. Kurdahi, A.C. Parker, "REAL: A Program for Register Allocation", *24th Design Automation Conf.*, 1987, pp. 210-215.

[13] F. Depuydt, "Register Optimization and Scheduling for Real-Time Digital Signal Processing Architectures", Ph. D. thesis, Katholieke Universiteit Leuven, Nov. 1993.

[14] D. Lanneer, M. Cornero, G. Goossens, H. DeMan, "Data Routing: a Paradigm for Efficient Data-path Synthesis and Code Generation", *Int. High-Level Synthesis Symposium*, May 1994, pp. 17-22.

[15] F. Franssen, L. Nachtergaele, H. Samsom, F. Catthoor, H. De Man, "Control flow optimizatioin for fast system simulation and storage minimization", *Eur. Design and Test Conf.*, 1994, pp. 20-24.

[16] W. Verhaegh, P. Lippens, E. Aarts, J. Korst, J. van Meerbergen, A. van der Werf, "Modelling Periodicity by PHIDEO Streams", *Int. Workshop on High-Level Synthesis*, 1992.