ACAP: Binary Synthesizer Based on MIPS Object Codes

Nagisa Ishiura
School of Science and Technology
Kwansei Gakuin University
2-1 Gakuen
Sanda, Hyogo, 669-1337, Japan
Email: nagisa.ishiura@ml.kwansei.ac.jp

Hiroyuki Kanbara
ASTEM RI
134 Chudoji Minamimachi
Shimogyo-ku, Kyoto, Japan 600-8813
Email: kanbara@astem.or.jp

Hiroyuki Tomiyama
College of Science and Engineering
Ritsumeikan University
1-1-1 Noji-Higashi
Kusatsu, Shiga 525-8577, Japan
Email: ht@fc.ritsumei.ac.jp

***Abstract*—This paper presents a binary synthesizer "ACAP," which synthesizes register transfer level HDL from MIPS object codes. It has three operation modes; (1) a separate compilation mode, in which selected subprograms of the target system are synthesized into hardware, (2) a full synthesis mode, where a whole linked executable code is transformed into a hardware module, and (3) an accelerator synthesis mode, where user specified sections of an executable code are converted into a hardware accelerator which is tightly coupled with the CPU. This paper describes several unique techniques adopted in ACAP, focusing on how interface between software and synthesized hardware is established.**

I. Introduction

Embedded systems are getting increasingly rich in functionalities. While large part of these functionalities are realized by software running on embedded processors, critical parts, for which real-time response or high-performance under limited power consumption is required, must be implemented as hardware. In order to meet the short time-to-market, efficient methodologies are needed for designing systems consisting of software and hardware.

High-level synthesis [1] is one of the most prevailing techniques for this purpose, where systems' specifications in high-level behavioral languages, such as C, SystemC, etc., are automatically compiled into a register transfer level hardware models which are ready for logic synthesis. Several methodologies have been proposed to expedite hardware/software codesign by utilizing high-level synthesis [2]–[4].

Binary synthesis [5] is a variant of high-level synthesis, in which input behavior is given in the form of object codes, instead of programs in high-level languages. The object codes may result from programs in some programming languages or from hand written assembly programs. Binary synthesis can handle wider range of software codes than high-level synthesis. Although the performance of the resulting hardware is generally lower than those obtained by high-level synthesis, due to lack of high-level information in the given programs, binary synthesis is attractive for the purpose of converting part of software into hardware.

This paper presents binary synthesizer "ACAP," which translates MIPS binary codes into synthesizable Verilog HDL descriptions. It aims at converting critical parts of programs into hardware which are originally intended to run on CPUs. ACAP implements several unique techniques for this purpose which take advantage of binary synthesis.

II. Binary Synthesis

Basic technologies for binary synthesis are not very much different from those for high-level synthesis. CDFGs (control dataflow graphs) are constructed from object codes instead of programs written in high-level languages. Almost all of the high-level synthesis techniques can be applied to the CDFGs to generate hardware.

The merit of binary synthesis is that it has less restrictions on input behavior specifications. For example, C programs with pointers or complicated control structures can be synthesized into hardware. Moreover, binary synthesizer may handle programs in multiple languages or hand-written assembly programs.

Mittal et al. [6] developed a binary synthesizer to translate DSP binaries into FPGA hardware. It could accept programs in C/C++, Matlab, and Simulink as well as hand-written assembly. The binary synthesizer developed by Stitt et al. [5] could synthesize selected sections of binary codes of MIPS, ARM, and MicroBlaze into coprocessors, or hardware accelerators. They also succeeded in enhancing the performance of the binary synthesizer to the same level of high-level synthesis by decompiling the given binary codes to recover high-level constructs. The synthesized accelerators were activated from the software running on the CPUs. Relatively simple interface was adopted, where arguments and return values were passed through global variables and the control was transferred by polling. The interface was created by *binary updating* which modifies the software binary code so that instructions for interface would be inserted and corresponding adjustment of instruction addresses would be done.

III. Binary Synthesizer ACAP

ACAP is a binary synthesizer that converts MIPS binary codes into hardware equivalent in terms of the behavior. Fig. 1 illustrates the synthesis flow of ACAP. It takes a MIPS binary code which may be generated by `gas` (an assembler) or `gcc` (a C compiler). It recovers assembly programs via `objdump` (a disassembler) and convert them into CDFGs. The conversion is based on a simple method which substitutes each instruction into a series of atomic operations. The synthesis engine performs optimization, scheduling, and binding, which are based on typical high-level synthesis algorithms, and generates Verilog HDL. We assume MIPS R3000 instruction set which handles 32bit addresses and data. ACAP is implemented in

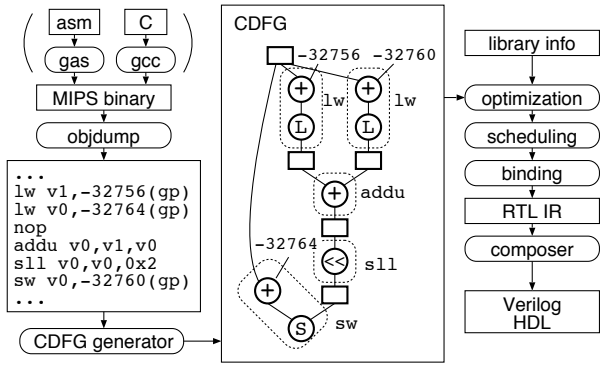

Fig. 1. Flow of synthesis in ACAP.

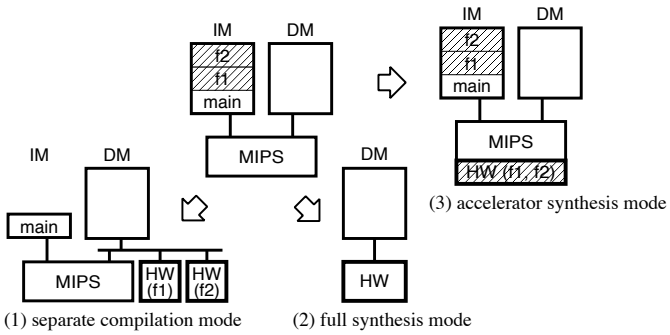

Fig. 2. Three synthesis modes of ACAP.

Perl5 and runs on Unix systems (including Cygwin and Mac OSX).

ACAP has three operation modes, as illustrated in Fig. 2, which generates different types of hardware from different types of inputs.

(1) Separate compilation mode

Subprograms chosen by users (`f1` and `f2` in Fig. 2) are synthesized into hardware (`HW(f1)` and `HW(f2)`, respectively) which work like the original subprograms; they share the memory space with the CPU and are callable from the software part. The programs are assumed to be written in C, though the synthesizer engine works on object codes, and the interface between software and hardware is arranged by source code level transformation on the input C programs.

(2) Full synthesis mode

A linked executable code is compiled into a hardware module whose behavior is equivalent to the code running on the CPU.

(3) Accelerator synthesis mode

Some selected sections in a linked executable code are synthesized into a hardware accelerator (`HW(f1,f2)` in the figure) which is tightly coupled with the CPU. The accelerator shares the register file as well as the main memory with the CPU and transfers the control and the data from/to CPU extremely quickly.

Both (1) and (3) aim at converting critical parts of programs, which are originally developed to run on CPUs, into hardware.

main.c

```
1  int v;
2  int f(int a);
3  int g(int x);
4
5  int main(void) {
6    ...
7    s = f(p+2);
8    t = g(s);
9    ...
10 }
```

(a) Main file

sub.c

```
1  extern int v;
2
3  int f(int a) {
4    ...
5    return r;
6  }
7
8  int g(int x) {
9    ...
10   return y;
11 }
```

(a) Sub file

sub_SW.c

```
1  extern int v;
2  volatile int _RUN_sub;
3  int _ARG_f_a, _RET_f;
4  int _ARG_g_x, _RET_g;
5  #define CALL_f 1
6  #define CALL_g 2
7
8  int f(int a) {
9    _ARG_f_a = a;
10   _RUN_sub = CALL_f;
11   while (_RUN_sub) { ; }
12   return _RET_f;
13 }
14
15 int g(int x) {
16   _ARG_g_x = x;
17   _RUN_sub = CALL_g;
18   while (_RUN_sub) { ; }
19   return _RET_g;
20 }
```

(c) Stub for sub

sub_HW.c

```
1  extern int v;
2  extern volatile int _RUN_sub;
3  extern int _ARG_f_a, _RET_f;
4  extern int _ARG_g_x, _RET_g;
5  #define CALL_f 1
6  #define CALL_g 2
7
8  int f(int a) {
9    ...
10   return r;
11 }
12
13 int g(int x) {
14   ...
15   return y;
16 }
17
18 nt main(void) {
19   for (;;) {
20     while (!_RUN_sub) { ; }
21     if (_RUN_sub == CALL_f) {
22       _RET_f = f(_ARG_f_a);
23     }
24     else if (_RUN_sub == CALL_g) {
25       _RET_g = f(_ARG_g_x);
26     }
27     _RUN_sub = 0;
28   }
29 }
```

(d) Input to syntesizer

main2.c

```
1  iint v;
2  int f(int a);
3  int g(int x);
4
5  int _main(void) {
6    ...
7    s = f(p+2);
8    t = g(s);
9    ...
10 }
11
12 int main(void) {
13   _INIT_sub();
14   return _main();
15 }
```

(e) Main for relinking

sub_SW2.c

```
1  extern int v;
2  void* volatile _RUN_sub;
3  int _ARG_f_a, _RET_f;
4  int _ARG_g_x, _RET_g;
5  #define CALL_f ((void*) 1)
6  #define CALL_g ((void*) 2)
7
8  void INIT_sub(void) {
9    void* volatile addr_table[]=
10   {
11     &v,
12     &_ARG_f_a, &_RET_f,
13     &_ARG_g_x, &_RET_g,
14   };
15   _RUN_sub = (void*) addr_table;
16   while(_RUN_sub){ ; }
17 }
18
19 int f(int a) {
20   _ARG_f_a = a;
21   _RUN_sub = CALL_f;
22   while (_RUN_sub) { ; }
23   return _RET_f;
24 }
25
26 int g(int x) { ... }
```

(f) Stub for relinking

sub_HW2.c

```
1  extern volatile int _RUN_sub;
2  #define CALL_f (void*) 1
3  #define CALL_g (void*) 2
4
5  int f(int a) {
6    ...
7    return r;
8  }
9
10 int g(int x) {
11   ...
12   return y;
13 }
14
15 int main(void) {
16   while (!_RUN_sub) { ; }
17   int *_v = *((int**) _RUN_sub + 0);
18   int *_arg_f_a = *((int**) _RUN_sub + 1);
19   int *_ret_f = *((int**) _RUN_sub + 2);
20   int *_arg_g_x = *((int**) _RUN_sub + 1);
21   int *_ret_g = *((int**) _RUN_sub + 2);
22   _RUN_sub = (void*) 0;
23
24   for (;;) {
25     while (!_RUN_sub) { ; }
26     if (_RUN_sub == CALL_f) {
27       *_ret_f = f(*_arg_f_a);
28     }
29     else if (_RUN_sub == CALL_g) {
30       *_ret_g = f(*_arg_g_x);
31     }
32     _RUN_sub = (void*) 0;
33   }
34 }
```

(g) HW functions for relinking

Fig. 3. Source code transformation for separate compilation mode.

IV. Separate Compilation Mode

A. *"Module per file" synthesis*

In the separate compilation mode, we assume that a target system is composed of multiple C program files, which are compiled to object codes, linked together as a binary program, and run on a CPU. Then, a user decides to convert some of the program files into hardware. Each file, which may contain single or multiple functions, is synthesized into a single hardware module.

While Stitt's method [5] needed binary level code modification to establish interface between software and hardware, ACAP does the equivalent task in source code level, leaving the complicated relocation tasks to a linker [8].

Fig. 3 shows an example of our code modification. A main file (a) is run on MIPS while a sub file (b), which contains functions `f` and `g`, will be synthesized into a hardware module. For this purpose, the sub file (b) is converted into two files (c) and (d), where the former is linked with the main file (a) and run as software while the latter is an input to the synthesizer.

The file (c) is a *stub* file; when function `f` is called from (a), the stub function `f` (in lines 8–13) is executed, which stores the argument to global variable `_ARG_f_a` and instruct the hardware to run the body of `f` by setting the function ID to `_RUN_sub`. On detecting the completion of hardware (when `_RUN_sub` becomes zero), it forwards the return value in `_RET_f` to the caller.

The file (d) is the hardware counterpart of (c). Shown in lines 19–28 is the main loop of the hardware behavior. It polls variable `_RUN_sub` and branches to the function specified in the variable.

Note that the three functions in (d) will be merged into a single hardware module during the synthesis [10]. A function call compiles into "`JAL` *address*" instruction. This is translated into an operation to save the return *state*, instead of the return address, into register `ra`, and an operation to set the next state to the one corresponding to the jump address. A return statement is translated to "`BR ra`" (branch to the address in `ra`), which is synthesized into an operation to alter the state variable. Passing of arguments and return values is converted into proper instruction sequences by a C compiler, so the binary synthesizer have only to translate them faithfully.

However, the addresses of the global variables must be known during synthesis of (d). They are acquired from the symbol table emitted by a linker when linking (a) and (c). The binary synthesizer patches in the addresses before logic synthesis.

B. Synthesis of relinkable hardware

In the method described in the previous subsection, logic synthesis must be re-run every time the addresses of global variables change due to modification on the software part, even though there is no change on the hardware part. For the same reason, the hardware must be resynthesized when one want to link the hardware with different main routines.

In ACAP, the software transformation method is further enhanced so that it can synthesize *relinkable* hardware, which can be relinked with modified software or different software without resynthesis [9]. This is achieved by passing an address table from software to hardware which contains the necessary global variables' addresses.

In Fig. 3, the main program (a) is transformed into (e); the main function calls initialization routine `INIT_sub` which creates the address table and passes it to the hardware. The hardware part (b) is transformed into (f) and (g). The address table is prepared in `INIT_sub` in (f) and received at the beginning of `main` in (g).

V. Accelerator Synthesis Mode

The accelerator synthesis mode compiles selected part of a binary code into a hardware accelerator *tightly coupled* with the CPU, where the transfer of the control and data between CPU and accelerator is extremely efficient [11], [12].

Fig. 4 shows how the synthesized accelerator works. The accelerator watches PC (via *a*) and starts execution as soon as the PC hits one of the the starting addresses of the hardware sections. While the accelerator is running, it deactivates CPU by feeding the same address to PC (via *b*) and NOP to IR (via *c*). The accelerator reads and writes the register file of the CPU directly or through the forwarding unit (via *d*, *e*, and *h*). It also accesses the same address space as that of CPU (via *f* and *g*). When the accelerator is *about* to finish its task, it writes the address to resume software execution into PC. Note that this PC write is done as early as possible to reload the instruction pipeline but not too early to interfere with the register/memory accesses of the remaining operations in the accelerator.

The input to this synthesis mode is a *linked* executable code. The sections to be synthesized are specified by users by inserting pragmas into the disassembled code. The synthesis method for this mode is very similar to the one in the previous section. However, register read operations (to get values from general purpose registers of the CPU) must be inserted to the hardware DFG at the point where the registers are first read in the hardware section. The same handling is necessary for register writes, at the end of the hardware section. However, if we do this insertion of register read/write operations DFG by DFG, it will results in many redundant operations. So we execute data flow analysis and delete unnecessary read/writes [14].

The strength of this accelerator is that switching between CPU and hardware is ultimately quick. There are no need for coprocessor instructions nor data transfer instructions, and the operations of the CPU and hardware are seamlessly executed.

The accelerator mode allows higher freedom in choosing the part of the programs to be synthesized. A hardware section should not necessarily be a function. It can be any consecutive sequence of instructions so that users can specify a loop or just a sequence of several instructions. The accelerator mode allows multiple sections as well as a single section and various types of control transfer to be synthesized into an accelerator [13]. It is possible to jump from hardware to hardware as well as from hardware to software. It is also possible to jump from the middle of a hardware section and jump into a hardware section. The accelerator can even call software functions. This is useful in processing data intensive tasks by hardware and letting software handle errors, or making system calls from hardware.

Linked executables, from which synthesis of this mode starts, include library codes such as string manipulation, multi-precision and soft float runtime libraries, etc., and they can also be synthesized. Thus, we could even accelerate floating point operations with a datapath equipped only with fixed point arithmetic units.

Another merit of this synthesis mode is that there is no need to alter binary codes nor need to modify compilers at all. Desired part of the executable code can be accelerated even if it resides in an ROM.

Experimental results of this synthesis mode are summarized in TABLE I. "AES" is an encryption program and the other two are from the CHStone [7]. In each benchmark program, the most frequently called function was synthesized into an hardware accelerator. The MIPS CPU consisted of 3,211 slices and its register to register delay was 26.0ns (as in

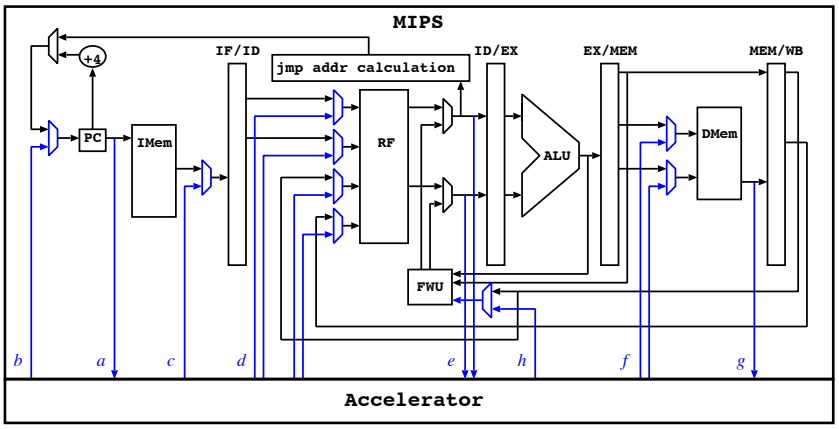

Fig. 4. Interaction between MIPS and accelerator [12].

TABLE I. SYNTHESIS RESULTS [14].

prog	HW	slice	cycle	delay [ns]
AES	MIPS only	3,221 (1.00)	47,953 (1.00)	26.0
	MIPS+ACC	4,643 (1.44)	32,353 (0.67)	28.6
SHA	MIPS only	3,221 (1.00)	746,649 (1.00)	26.0
	MIPS+ACC	7,642 (2.37)	250,125 (0.33)	42.9
Blowfish	MIPS only	3,221 (1.00)	761,878 (1.00)	26.0
	MIPS+ACC	7,310 (2.27)	375,448 (0.49)	62.3

The target was Xilinx Spartan-3E FPGA and the synthesizer was Xilinx ISE 12.4.

the "MIPS only" rows). Since the delay of an ALU was about 7.0ns, three level chaining was performed using 9 ALUs. As shown in the "MIPS+ACC" rows, the accelerators reduced the total execution cycles into 33% to 67% at the cost of hardware increase by 1.44 to 2.27 times. Although the delays were much larger than that of MIPS, all of them were confirmed to be from false paths which would never be activated and the actual delay would be as much as 26 to 30ns.

VI. CONCLUSION

Some technical features of ACAP has been introduced focusing on how interface between software and synthesized hardware is established. Binary synthesis seems to be a promising tool to bridge between software and hardware. Currently, ACAP is not very good at performance in terms of the speed and area of synthesized hardware. We are continuingly working on improving the usability and performance of ACAP.

ACKNOWLEDGMENT

We would like to express our thanks to all the people who have contributed to develop ACAP; Mr. Tatsuya Ikegami, Mr. Yoshitaka Iritani, Mr. Yuki Toda, Mr. Makoto Orino, Mr. Fumiaki Takashima, Mr. Shunsuke Satake, Mr. Shimpei Tamura, who were with Kwansei Gakuin University, Mr. Naoya Ito, Mr. Hinata Takebayashi, who are with Kwansei Gakuin University, Mr. Takayuki Nakatani who were with Ritsumeikan University, and Masaharu Yano, who were with Kyoto University.

REFERENCES

[1] D. D. Gajski, N. D. Dutt, A. C-H Wu, and S. Y-L Lin: *High-Level Synthesis: Introduction to Chip and System Design,* Kluwer Academic Publishers (1992).

[2] S. L. Shee, S. Parameswaran, and N. Cheung: "Novel architecture for loop acceleration: A case study," in *Proc. CODES+ISSS '05*, pp. 297–302 (Sept. 2005).

[3] S. Shibata, S. Honda, H. Tomiyama, and H. Takada: "Advanced System-Builder: A tool set for multiprocessor design space exploration," in *Proc. ISOCC 2010*, pp. 79–82 (Nov. 2010).

[4] K. Seto and M. Fujita: "Custom instruction generation for configurable processors with limited numbers of operands," in *IPSJ Trans. on System LSI Design Methodology*, vol. 3, pp. 57–68 (Feb. 2010).

[5] G. Stitt and F. Vahid: "Binary synthesis," *ACM Trans. on Design Automation of Electronic Systems*, vol. 12, no. 3, article 34 (Aug. 2007).

[6] G. Mittal, D. C. Zaretsky, X. Tang, and P. Banerjee: "Automatic translation of software binaries onto FPGAs," in *Proc. DAC 2004*, pp. 389–394 (June 2004).

[7] Y. Hara, H. Tomiyama, S. Honda, H. Takada, and K. Ishii: "CHStone: A benchmark program suite for practical C-based high-level synthesis," in *Proc. ISCAS 2008*, pp. 1192–1195 (May 2008).

[8] (in Japanese) Y. Iritani, T. Ikegami, N. Ishiura, H. Kanbara, and H. Tomiyama: "Implementation of a high-level synthesis system which uses MIPS assembly programs as intermediate representation," in *IPSJ SIG Technical Report*, 2010-SLDM-144-58 (Mar. 2010).

[9] (in Japanese) M. Orino, N. Ishiura, H. Tomiyama, F. Takashima, and H. Kanbara: "High-level synthesis of hardware relinkable to software," in *Technical Report of IEICE*, VLD2011-107 (Jan. 2012).

[10] (in Japanese) F. Takashima, N. Ishiura, M. Orino, H. Tomiyama, and H. Kanbara: "Merge of functions in high-level synthesis using assembly codes as intermediate representation," in *Technical Report of IEICE*, VLD2011-106 (Jan. 2012).

[11] (in Japanese) Y. Toda, N. Ishiura, H. Kanbara, and H. Tomiyama: "Hardware/software co-design based on coprocessor tightly coupled with CPU," in *IPSJ SIG Technical Report*, 2010-EMB-15-16 (Jan. 2010).

[12] (in Japanese) N. Ito: *Fast Execution switching between CPU and Hardware Accelerators*, Bachelor Thesis, School of Science and Technology, Kwansei Gakuin University (Mar. 2014).

[13] (in Japanese) S. Satake, N. Ishiura, S. Tamura, H. Kanbara, H. Tomiyama: "Speeding up multiple sections of binary code by hardware accelerator tightly coupled with CPU," in *Technical Report of IEICE*, VLD2012-119 (Jan. 2013).

[14] (in Japanese) S. Tamura, N. Ishiura, H. Kanbara, H. Tomiyama: "Binary synthesis of hardware accelerator tightly coupled with CPU," in *Technical Report of IEICE*, VLD2013-133 (Jan. 2014).