Power-efficient Instruction Encoding Optimization for Various Architecture Classes

D. Zhang, A. Chattopadhyay, D. Kammler, E. M. Witte
G. Ascheid, R. Leupers, H. Meyr
Institute for Integrated Signal Processing Systems
RWTH Aachen University, Germany
zhang@iss.rwth-aachen.de

***Abstract*— A huge application domain, in particular, wireless and handheld devices strongly requires flexible and power-efficient hardware with high performance. This can only be achieved with Application Specific Instruction-Set Processors (ASIPs). A key problem is to determine the instruction encoding of the processors for achieving minimum power consumption in the instruction bus and in the instruction memory. In this paper, a framework for determining power-efficient instruction encoding in RISC and VLIW architectures is presented. We have integrated existing and novel techniques in this framework and propose novel heuristic approaches. The framework accepts an existing processor's instruction-set and a set of implementations of various applications. The output, which is an optimized instruction encoding under the constraint of a well-defined cost model, minimizes the power consumption of the instruction bus and the instruction memory. This results in strong reduction of the overall power consumption. Case studies with commercial embedded processors show the effectiveness of this framework.**

***Index Terms*— power-efficient, instruction encoding, instruction memory, instruction bus, embedded processors**

I. Introduction

Due to the unique blend of performance and flexibility, application-specific processors are used increasingly as components of modern complex System-on-Chips (SoCs). The design of such application-specific processors remains a huge challenge owing to the conflicting design goals such as power, performance, flexibility etc. During the past years, two major design methodologies came up to aid the processor designer. The first one targets a processor design from scratch using a high-level Architecture Description Language (ADL) [1] [2] [3]. The second methodology extends or customizes a template processor [4] [5]. For both kinds of processor design approaches, the instruction encoding plays a major role in determining the power consumption of the system [6].

The existing instruction encoding synthesis methods [7] [8] during embedded processor design, do not cover the effect of coupling capacitance and therefore are inadequate for deep submicron technologies. In [9] and [10], a low-power instruction encoding approach is described considering both self and coupling capacitances. This technique is applied after the processor is designed and for that reason an external hardware is used to decode the instruction. In this paper, a methodology for detecting the instruction-set encoding during the processor design phase is proposed, which does not require external hardware. Furthermore, we show that the framework used in our approach can also be applied to only change the assembly instructions to minimize power, without any changes in the hardware.

This Paper is based on "Power-efficient Instruction Encoding Optimization for Embedded Processors," by A. Chattopadhyay, D. Zhang, D. Kammler, E. M. Witte, R. Leupers, G. Ascheid, and H. Meyr, which appeared in the VLSI Design Conference, Bangalore, India, January 2007.

A. Instruction Memory Power Consumption

Published results [7] on the power distribution of different processor cores reveal that the instruction memory power consumption can have a strong impact on the overall processor power. This can be seen, for example, in the ICORE ASIP, where instruction ROM contributes up to 32.4% of total processor power (figure 1).

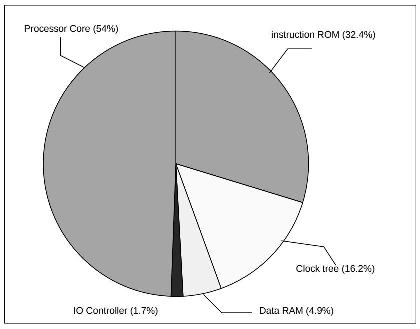


Figure 1. Power Distribution for ICORE

The toggling activity in the instruction word has direct influence on the instruction memory power consumption. Exemplarily, NMOS-ROMs use a two-phase access scheme, where during the first phase the bit lines are precharged and during the second phase row decoder asserts one word line. If a specific bit cell contains logic 0, then the associated bit line is discharged, therefore resulting in power dissipation. According to the case study in [11], 70% of the total energy consumption of an SRAM is required for the bit lines, the associated sense amplifiers and the bit cells themselves. In order to reduce the memory power consumption, it would suffice to reduce the toggling activity of the adjacent instructions. However, the instruction bus power consumption demands a more generic power model as presented below. This power model is used for determining the optimized instruction encoding.

B. Instruction Bus Power Consumption

An embedded processor typically accesses on-chip or off-chip instruction memory via a bus. Two effects are relevant for the power consumption in the bus lines. The first effect is caused by the *self capacitance* of the bus lines. The charging or discharging of this self capacitance occurs due to the toggling of bus lines. The second contribution to the power consumption results from the *coupling capacitance* of adjacent bus lines. Coupling capacitance is more prominent for technologies below 0.25 μm and for long off-chip buses, where the bus lines are conventionally routed close to each other. With this perspective, it becomes important to consider not only the toggling of subsequent bits but also the toggling of adjacent bits in the bus lines [12]. Consequently, detailed modelling and power analysis for considering the effects of opcode space and crosstalk has been done in recent years [13]. The power model, which is used in this paper, is described in the following.

Considering only $0 \rightarrow 1$ transitions as the power-consuming transitions, which charges the capacitances, the *power consumption* can be written as:

$$P_{self} = \alpha f C_s {V_{dd}}^2 \tag{1}$$

Here α is the average number of $0 \rightarrow 1$ bit transitions, f is the clock frequency, C_s is the self capacitance of the bus line and V_{dd} is the supply voltage. This model has been extended in [13] [14], where the power consumption due to the coupling capacitance (C_c) is also considered. This power consumption arises from following different transitions between adjacent bus lines.

- transition type 1: only one of the two lines toggle and the final values of the adjacent bit lines are different e.g. $00 \rightarrow 01$. The average number of such transitions is referred as β.
- transition type 2: both of the lines toggle to different final values e.g. $01 \rightarrow 10$. The average number of such transitions is referred as γ.

Transition type 2 causes the coupling capacitance to switch twice, while the other type of transition causes it to switch once. Assuming $\lambda = \frac{C_c}{C_s}$, which changes with technology, the overall power consumption can be written as following. The value of λ can be up to 3 for 0.18 μm technology [13].

$$\begin{aligned} P_{con} &= P_{self} + P_{coupling} \qquad (2) \\ &= \alpha f C_s {V_{dd}}^2 + (\beta f C_c {V_{dd}}^2 + 2\gamma f C_c {V_{dd}}^2) \\ &= {V_{dd}}^2 f C_s(\alpha + \beta\lambda + 2\gamma\lambda) \end{aligned}$$

For the work in this paper, we have varied λ from 0 to 4, with higher values indicating deeper sub-micron technology.

The rest of the paper is organized as follows: section II introduces the previous work in this domain and outlines the contribution of this paper. Section III describes the overall optimization framework for instruction encoding synthesis. In section IV and V, the optimization algorithms are elaborated in detail. The results are analyzed in section VI. This paper ends with a summary and an outlook.

II. Related Work

The minimization of power consumption in a processor became a key research topic with increasing system complexity and shrinking power budget. At the level of physical design, there exist techniques for reducing power consumption in deep submicron buses [12] [15]. Complementing these, there exist power-aware instruction encoding synthesis methods during processor design.

The power-aware instruction encoding optimization techniques can be classified into two categories according to the part of the instruction it deals with. The hardware-oriented techniques (which are applicable during new processor design) target the opcode. The software-oriented techniques deal with the encoding of the operands of the instruction. The latter kind of techniques can be applied even after the processor hardware is implemented.

Through hardware approach, the opcode of an instruction is modified to minimize power consumption. This requires subsequent updating/generation of the processor decoder [7] [16] or addition of external hardware [9]. Understandably, these techniques cannot be easily applied to the extensible processor design methodology, where the hardware is fixed beforehand. In [7] a method is outlined to obtain power-efficient opcodes during processor design. It assigns the maximum weighted code word in a greedy manner to the most frequently occurring pair of instructions, thereby avoiding power-dissipating discharging in the memory bit-lines. This method did not consider any coupling effect. Similarly, the approach taken at [8] aims at a reduction of the hamming distance of opcode between the most frequently occurring pairs of instructions. In [9], the complete instruction word is transformed dynamically using an external hardware coupled with the processor's fetch unit. Essentially, this method can be considered as a variant of techniques presented in [15] applied during physical design. These approaches have high flexibility, because they can be applied to arbitrary instruction-sets. Any part of the complete instruction word can be chosen for encoding. This offers strong optimization potential. The major drawback of these approaches is that it is employed after the processor is completely designed and therefore, the achievable low power optimization may conflict with other performance metrics at a very late design phase.

Our approach does not require additional hardware, rather the instruction decoder has to be adapted to the modified instruction-set, which saves power and is complementary to the approach presented in [10] and an extension of the approaches presented in [7].

In software-based approaches, a post-assembly optimization is plugged in for minimizing power-sensitive transitions in the instruction word. These optimizations do not call for any additional external decoder. For example, the technique presented in [9] utilizes Register Name Adjustment (RNA) in order to rename the register operands in the assembly program using a greedy algorithm. As will be shown in this paper, a heuristic solution for RNA improves the result considerably. Furthermore, in [9] the RNA algorithm is applied within the program hot-spots aided by a profiling tool. We show that using a graph-based data structure, a hash table based data structure and high level simulation, the transition information for the entire program can be obtained and utilized precisely.

In summary, for the hardware-oriented techniques the sophisticated power models are not used during instruction encoding at the processor design phase and the software-oriented techniques are sub-optimal. This paper contributes in both of these areas.

- Firstly, we offer a framework, which accepts an existing processor instruction-set and a group of assembly programs. The output is an optimized instruction encoding under the constraints of a well-defined cost model, which minimizes the power consumption for the target group of programs.
- Secondly, we present effective heuristic algorithms to minimize memory and bus power consumption under a given power model.

III. Instruction Encoding Synthesis Flow

In this section, the overall instruction encoding synthesis flow is outlined. On the basis of the flow, the optimization problem is formulated.

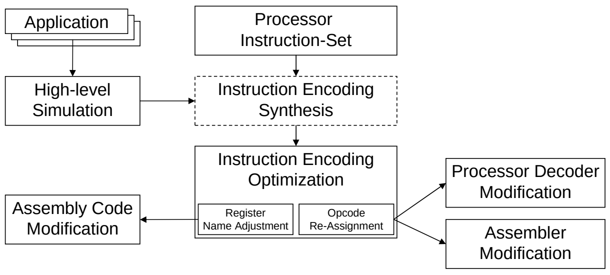

Application
Processor Instruction-Set
High-level Simulation
Instruction Encoding Synthesis
Instruction Encoding Optimization
Register Name Adjustment
Opcode Re-Assignment
Assembly Code Modification
Processor Decoder Modification
Assembler Modification

Figure 2. Overall Encoding Synthesis Flow

A. Overall Encoding Synthesis Flow

The encoding synthesis flow for application-specific processors is depicted in figure 2. At first, the applications are simulated using high-level simulators [17]. The information is used to determine the optimum instruction width requirements. This includes the space requirements for opcode, immediate bits, register operands etc. This phase is referred as *instruction encoding synthesis*. Traditionally, the transition information is neglected in the instruction encoding synthesis, as it would highly increase the problem complexity. The approaches, which considered the transition information [7] [8], do not consider the cross-coupling effect.

The work presented in this paper, deals exactly with this problem. In this paper an initial instruction encoding is taken and optimized according to an enhanced power model. As shown in figure 2, the instruction encoding can be optimized by performing Opcode Re-Assignment (ORA) and/or Register Name Adjustment (RNA). For ORA, the encoding of the opcode elements of each instruction is modified. Therefore, it requires the modification of the processor decoder and also the assembler. Usually, these are generated automatically in an ADL-based ASIP design framework [1] [2]. The other technique, originally presented in [9], performs a renaming of the registers in the hot spots of the assembly program. The register name adjustment, without altering the program functionality, alters the bit-transition characteristics of the overall program. This reduces the power consumption in the instruction bus as well as in the instruction memory.

In this paper the overall tool-flow is applied to an ADL-based ASIP design environment. The ORA technique modifies the instruction encoding in the ADL. From the ADL description of the processor, the assembler, linker and the RTL implementation including decoder are automatically generated. The RNA technique basically is a one-to-one mapping of each register in the assembly program. A shell-script running over the assembly program can modify it easily.

B. Problem Formulation

Opcode Re-Assignment (ORA) : Consider a total of M instructions, where each instruction (i) is having a binary opcode of $C(i)$. Our goal is to determine a new set of $C(j)$ such that the power consumption P_{con} is minimized and $C(i) \neq C(j)$, $\forall i, j \epsilon M$, where $i \neq j$.

Register Name Adjustment (RNA) : Consider a total of N allocatable registers, where each register (reg_i) is having a binary coding of $C(reg_i)$. Our goal is to find a bijective mapping $B_{RNA} : reg_i \rightarrow reg_j$, such that P_{con} is minimized.

IV. Algorithmic Outline of the Solution

In this section, the algorithmic outline of our proposed solution is described. Initially, the processor instruction-set, which is to be optimized, is represented using a *grammar file*. Using the grammar file and the assembly program, a graph-like data structure and a hash table based data structure are constructed, which store the information about the opcode and register elements. They also store the transition information, loaded from pre-performed simulation of the program. Finally, based on both data structures, the formulation for P_{con} is done and the algorithms for ORA and RNA are outlined. An extension of this approach for optimization based on multiple assembly programs will be shown in section V.

A. Grammar File Format

The instruction grammar represents the valid instructions in Backus-Naur Form (BNF) grammar. Table I shows an exemplary *instruction grammar*. For this example, the instruction word width is 16 bit and there are 16 available registers indexed by `src` and `dst`. The opcodes are given in terminals '0' and '1', while terminals 'r', 'i' and 'x' represent registers, immediates and don't cares respectively. Syntactic variables in the grammar file are referred as non-terminals, for example "add", "sub", etc.

TABLE I.
EXEMPLARY INSTRUCTION GRAMMAR

insn	: add dst src src ‖ sub dst src src ‖ ld dst src imm ‖ nop ‖ jmp cond_src dst_imm
add	: 0001
sub	: 0010
jmp	: 0011
ld	: 10
src	: rrrr
cond_src	: rr
dst_imm	: iiiiiiiii
imm	: iiiiii
dst	: rrrr
nop	: 01xx 0000 0000 xxxx

B. Instruction Encoding Representation

From the instruction grammar file, a set of all possible instruction patterns of the processor is derived. Such a set for the abovementioned grammar file is represented in the table II.

TABLE II.
ALL POSSIBLE INSTRUCTION PATTERNS

insn_0(ld)	: 10rr rrrr rrii iiii
insn_1(add)	: 0001 rrrr rrrr rrrr
insn_2(sub)	: 0010 rrrr rrrr rrrr
insn_3(jmp)	: 0011 rrii iiii iiii
nop	: 01xx 0000 0000 xxxx

This instruction pattern set can be used as an instruction filter. The instructions in the assembly program can be compared with these instruction patterns to extract concrete information contained in them. The extracted information includes opcodes, registers, don't cares, immediates and μ-opcodes. A μ-opcode is a part of an opcode, which will be introduced in the following section in detail.

However, for VLIW architectures it is not reasonable to generate all possible instruction patterns. There are mainly two reasons for this. First, in a VLIW architecture the number of all instruction patterns is huge. Considering a five-slot VLIW architecture, even if only one slot contains 100 instruction patterns, the total number of instruction patterns of the architecture is 100^5 as the result of permutation of the instruction patterns in all slots. Second, since the decoding of instructions is independent from each other in different slots, it is not necessary to generate all the combinations between them. It is sufficient only to generate instruction patterns for different slots separately. Table III and table IV show an exemplary VLIW grammar file and the corresponding instruction patterns in different slots.

For extracting the information from the assembly program, the instructions only need to be compared with the instruction patterns slot by slot at corresponding bit-fields.

TABLE III.
EXEMPLARY INSTRUCTION GRAMMAR IN VLIW

root	: slot1 slot2 slot3
slot1	: insntype1
slot2	: insntype1 ‖ insntype3
slot3	: insntype2 ‖ insntype3
insntype1	: add dst src src ‖ sub dst src src
insntype2	: ld dst src imm ‖ nop
insntype3	: jmp cond_src dst_imm
add	: 0001
sub	: 0010
jmp	: 0011
ld	: 10
src	: rrrr
cond_src	: rr
dst_imm	: iiiiiiiii
imm	: iiiiii
dst	: rrrr
nop	: 01xx 0000 0000 xxxx

TABLE IV.
ALL POSSIBLE INSTRUCTIONS PATTERN IN SLOTS OF VLIW

slot1	insn_1(add)	0001 rrrr rrrr rrrr
	insn_2(sub)	0010 rrrr rrrr rrrr
slot2	insn_1(add)	0001 rrrr rrrr rrrr
	insn_2(sub)	0010 rrrr rrrr rrrr
	insn_3(jmp)	0011 rrii iiii iiii
slot3	insn_0(ld)	10rr rrrr rrii iiii
	insn_3(jmp)	0011 rrii iiii iiii
	nop	01xx 0000 0000 xxxx

C. ORA Optimization

For the ORA optimization, the position information of the opcodes in the instruction words is required. This can be obtained from the instruction grammar file. With this information, the toggling information and coupling information between an opcode and other opcodes and operands can be determined. Based on these informations, the opcodes can be re-assigend with new bit patterns, aiming at saving power.

Since the opcodes are used to identify different instructions, it is important to re-assign the opcodes in such a way, that the instructions can still be decoded unambiguously. For regular cases, where the opcodes lie in the same bit-field of the instruction words, as illustrated in figure 3(a), the problem can be simply solved by assigning the opcodes with different bit patterns. However, usually the construction of the opcodes is irregular, as shown in figure 3(b) and 3(c).

insn0:100xxxxx	insn0:100xxxxx	insn0:100xxxxx
insn1:001xxxxx	insn1:00111xxx	insn1:00100xxx
insn2:110xxxxx	insn2:11011xxx	insn2:00101xxx
(a)	(b)	(c)

Figure 3. Opcodes Example

In figure 3(b), instruction *insn0* has a different opcode bit-width from *insn1* and *insn2*, which means that the opcode of *insn0* is always different from that of *insn1* and *insn2*. However this is not sufficient for a unique decoding, because some bits of opcode of *insn1* and *insn2* are covered by the operand part of *insn0*. These bits in the operand part cannot be used to distinguish *insn0* from *insn1* and *insn2*, because operands (denoted by 'x') can have arbitrary bit patterns. To achieve a unique decoding in this case, the opcode of *insn0* must be assigned differently from the first three bits of the opcode in *insn1* and *insn2*.

Often the decoder structure in the processors is hierarchically organized. In the example in figure 3(c), the first three bits “001” of *insn1* and *insn2* indicate that both instructions might be grouped into a same instruction type. They are used to distinguish this instruction type from other types, in this case *insn0*. Then in this type a sub-decoder is used to distinguish *insn1* from *insn2* with the remaining two opcode bits. To assign the opcode of these three instructions, the first three bits of *insn0* must be assigned differently from those of *insn1* and *insn2*, and the first three bits of *insn1* and *insn2* must be assigned with the same coding, while the remaining two opcode bits of both instructions have to be assigned differently.

For the abovementioned reasons, the instruction patterns generated from the instruction grammar file are separated into several columns, in order to maintain the uniqueness of the opcode allocation in different branches of a hierarchical instruction-set. After that, column graphs and a hash table are created for the assembly program. Based on the column graphs and the hash table, a heuristic approach is used to optimize the assignment of the opcodes.

1) Column Separation for RISC: The column separation procedure is done in two steps.

- In the first step, for each possible instruction, a dividing line between the opcode bit-field and the other bit-fields is drawn.
- In the second step, the dividing line of each row is extended across the complete table.

Table V shows the modified instruction table of table II after the column separation.

TABLE V.
COLUMN SEPARATION

	col_1	col_2	col_3	col_4
insn_0	10	rr	rrrr rrii	iiii
insn_1	00	01	rrrr rrrr	rrrr
insn_2	00	10	rrrr rrrr	rrrr
insn_3	00	11	rrii iiii	iiii
nop	01	xx	0000 0000	xxxx

In this modified table, some columns capture a part of the opcode, which is referred as *μ-opcode*. Each *μ-opcode* contains the information about its position in the instruction and its bit-width. Important is, that the *μ-opcodes* with the same binary value in the same column are considered as the same *μ-opcode*, even if they may belong to different opcodes. By assigning different *μ-opcodes* in a same column with different bit patterns, the hierarchy of the coding architecture is maintained. This also results in the benefit of reduced complexity for finding a conflict-free instruction encoding, since it is sufficient to separately ensure unique bit pattern for the *μ-opcodes* of each column. Now the relation between Instructions and *μ-opcodes* can be derived from table V, which is shown in table VI.

TABLE VI.
INSTRUCTION PATTERNS AND μ-OPCODES

	col_1	col_2	col_3	col_4
insn_0	μ-op0	rr	rrrr rrii	iiii
insn_1	μ-op1	μ-op3	rrrr rrrr	rrrr
insn_2	μ-op1	μ-op4	rrrr rrrr	rrrr
insn_3	μ-op1	μ-op5	rrii iiii	iiii
nop	μ-op2	xx	μ-op6	xxxx

2) Column Separation for VLIW: For a VLIW architecture the separation of instruction patterns is more complicated. Since the decoding of instructions in different slots of a VLIW architecture is independent of each other, the assignment of opcodes in different slots in principle can also be done differently for each slot. For example in table IV, the opcode for “add” may be “0001” in slot1, while in slot2 it may be assigned with “0011”.

However, conventionally the same opcode in different slots is still assigned with the same bit pattern, because that simplifies the design of the instructions and also the development of assembler and disassembler greatly. Since the same opcode in different slots might be separated differently, the way of separating instruction patterns described above still needs to be extended. Without loss of generality, with the VLIW grammar file and instruction patterns in table III and IV, the separation of the instruction patterns for VLIW architectures is shown exemplarily in the following steps:

- The first step is to do column separation in different slots separately. An example is shown in table VII.

TABLE VII.
COLUMN SEPARATION IN SLOTS

slot1	insn_1	0001		rrrr rrrr rrrr	
	insn_2	0010		rrrr rrrr rrrr	
slot2	insn_1	0001		rrrr rrrr rrrr	
	insn_2	0010		rrrr rrrr rrrr	
	insn_3	0011		rrii iiii iiii	
slot3	insn_0	10	rr	rrrr rrii	iiii
	insn_3	00	11	rrii iiii	iiii
	nop	01	xx	0000 0000	xxxx

- In a second step, the separation positions of the identical opcodes in different slots have to be merged. This might produce new separation positions for other opcodes. Then these new generated separation positions need to be merged again. Thus, this procedure is recursive and settles when no more new separation positions are generated. For example, in table VII the opcode of “jmp” (coding 0011) in insn_3 is separated in slot3, but not in slot2. So the opcode of “jmp” in slot2 has to be separated, which introduces new separation positions in the opcodes of “add” (coding 0001) and “sub” (coding 0010) in slot2. This again requires the opcodes of “add” and “sub” in slot1 to be separated, too.
  At the end of this recursive procedure, the columns in the slots are updated, which are shown in table VIII.
- In a third step, *dependent columns* in different slots are determined. Dependent columns are the columns, which contain common bit-fields of certain opcodes.

TABLE VIII.
UPDATED COLUMNS IN SLOTS

		col_1	col_2	col_3	col_4
slot1	insn_1	00	01	rrrr rrrr	rrrr
	insn_2	00	10	rrrr rrrr	rrrr
		col_5	col_6	col_7	col_8
slot2	insn_1	00	01	rrrr rrrr	rrrr
	insn_2	00	10	rrrr rrrr	rrrr
	insn_3	00	11	rrii iiii	iiii
		col_9	col_10	col_11	col_12
slot3	insn_0	10	rr	rrrr rrii	iiii
	insn_3	00	11	rrii iiii	iiii
	nop	01	xx	0000 0000	xxxx

TABLE IX.
μ-OPCODES IN SLOTS

		col_1	col_2	col_3	col_4
slot1	insn_1	μ-op0	μ-op3	rrrr rrrr	rrrr
	insn_2	μ-op0	μ-op4	rrrr rrrr	rrrr
		col_5	col_6	col_7	col_8
slot2	insn_1	μ-op0	μ-op3	rrrr rrrr	rrrr
	insn_2	μ-op0	μ-op4	rrrr rrrr	rrrr
	insn_3	μ-op0	μ-op5	rrii iiii	iiii
		col_9	col_10	col_11	col_12
slot3	insn_0	μ-op1	rr	rrrr rrii	iiii
	insn_3	μ-op0	μ-op5	rrii iiii	iiii
	nop	μ-op2	xx	μ-op6	xxxx

For example, in table VIII col_1, col_5 and col_9 are dependent columns, because col_1 and col_5 have the same bit-field of the opcodes of "and" and "sub", and col_5 and col_9 have the same bit-field of the opcode of "jmp". The reason to determine dependent columns is that in a VLIW architecture the common bit-fields of an opcode in different columns still have to be assigned with the same bit patterns. By assigning same bit patterns to the common opcode bit-fields in dependent columns, a unique assignment of the opcodes is ensured.
The determination of dependent columns can be explained using a matrix representation, as shown in figure 4(a). The columns, which can be connected by horizontal lines at the crossing points are considered as dependent columns. In figure 4(b), an example for a set of dependent columns is shown, namely, $\{c_1, c_3, c_4\}$. The other dependent columns in this matrix are $\{c_2\}$ and $\{c_5, c_6\}$.

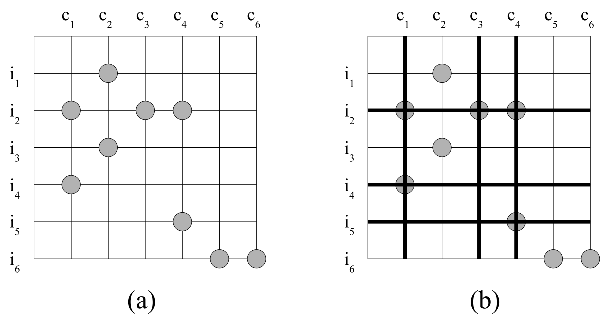


Figure 4. Example for Dependent Columns

- The last step is to assign the μ-opcodes to the bit patterns in the columns, which is similar to the assignment of μ-opcodes for RISC architectures described above. The only difference is that here same bit patterns in dependent columns must be referred as the same μ-opcode, instead of in a single column. The result after the assignment of the μ-opcodes is given in table IX.

3) Information Extraction in ORA: To extract the toggling and coupling information from the assembly program, the program is at first simulated and a sequence of instructions is dumped, which for example includes unrolled loops, etc. Then this instruction sequence is analysed and mapped to μ-opcodes. This mapping can be considered as "column-wise disassembling". Each instruction is compared with the instruction patterns, and only one instruction pattern will be met. Through the relation between the pattern and the μ-opcodes, information about the opcode for the instruction is extracted in form of corresponding μ-opcodes information. The *don't care*, *immediate* and *register* fields are instantiated with the appropriate values in the instruction. An example for the ORA information extraction is illustrated in figure 5.

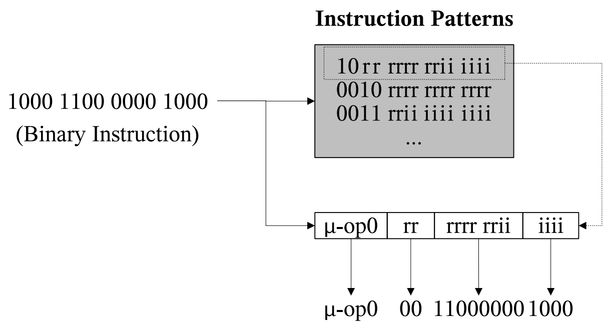


Figure 5. ORA Information Extraction

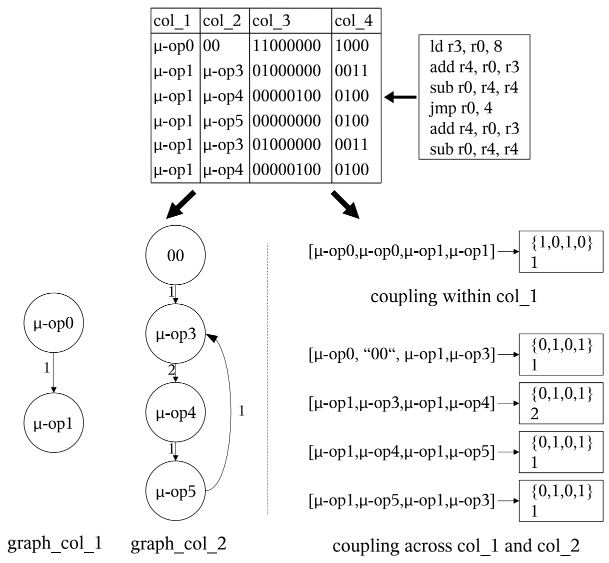
As shown in the figure, through comparing the incoming instruction word "1000110000001000" (assembly code: ld r3, r0, 8) with the instruction patterns, the pattern "insn0: 10rr rrrr rrii iiii" is matched (see table II). For the pattern *insn0*, the corresponding μ-opcode information can be obtained from table VI. So μ-op0 is extracted in the first column, and the information in the remaining columns is extracted by replacing the remaining parts of the pattern with the coding of the instruction word: "00", "11000000" and "1000".

With the extracted information about μ-opcodes, a directed graph for the toggling information in each column and a hash table data structure for the coupling information within columns and across columns can be generated.

4) ORA Column Graphs: From the extracted information, a *directed graph* for each column is prepared for the ORA. The existing μ-opcodes of the same binary value, are mapped to the same node of the graph.

Formally, the graph is defined as $\langle V, E \rangle$, where V represents an unique μ-opcode for the column and E is the edge set between two vertices. The edge set consists of two directed edges (E_1, E_2), each representing the frequency of occurrence of the adjacent nodes in one direction. The *column graphs* of col_1 and col_2 for a given assembly program are shown in figure 6. It can be observed that for col_1, there are two distinct binary values, resulting into two different μ-opcodes. There is only one edge between the two μ-opcodes, which represents the transition from the first instruction to the second instruction. In col_2, transition from μ-op3 to μ-op4 happens twice, therefore the correponding edge in the graph is weighted with 2.

In a VLIW architecture, single graphs are at first created for each column. Then the graphs of those columns, which belong to the same set of dependent columns, have to be combined into one graph to ensure a unique coding assignment for the μ-opcode-nodes.



col_1	col_2	col_3	col_4
μ-op0	00	11000000	1000
μ-op1	μ-op3	01000000	0011
μ-op1	μ-op4	00000100	0100
μ-op1	μ-op5	00000000	0100
μ-op1	μ-op3	01000000	0011
μ-op1	μ-op4	00000100	0100

```
ld r3, r0, 8
add r4, r0, r3
sub r0, r4, r4
jmp r0, 4
add r4, r0, r3
sub r0, r4, r4
```

00
μ-op0
μ-op1
μ-op3
μ-op4
μ-op5
1
2
1
1
1
graph_col_1
graph_col_2
[μ-op0,μ-op0,μ-op1,μ-op1]
{1,0,1,0} 1
coupling within col_1
[μ-op0, "00", μ-op1,μ-op3]
{0,1,0,1} 1
[μ-op1,μ-op3,μ-op1,μ-op4]
{0,1,0,1} 2
[μ-op1,μ-op4,μ-op1,μ-op5]
{0,1,0,1} 1
[μ-op1,μ-op5,μ-op1,μ-op3]
{0,1,0,1} 1
coupling across col_1 and col_2

Figure 6. ORA Column Graphs and Coupling Information

5) ORA Hash Table: As shown in figure 6, the coupling informaton among the nodes can also be created, while the column graphs are created.

The coupling information is evaluated both inside of the columns and across the columns. In the figure, only the coupling information within col_1 and between col_1 and col_2 is shown, which includes exact information, at which bits of the nodes a coupling transition occurs and how frequently it occurs. Inside of col_1, only one coupling transition occurs between the first bit and the second bit of μ-op0 and μ-op1. Across col_1 and col_2, coupling effects occur between the last bits of col_1 and the first bits of col_2. For example, a coupling transition happens twice from bit 0 of μ-op1 and bit 1 of μ-op3 to bit 0 of μ-op1 and bit 1 of μ-op4.

Our goal is to determine a new coding for each μ-opcode-node, such that the sum of P_{self} and $P_{coupling}$ within columns and $P_{coupling}$ across columns is minimized. Within a column, the only restriction is to have unique binary coding for each node. Across columns, there is no such restriction.

6) Solution of ORA: The task of determining the optimum encoding with the given constraints is an NP-complete problem, because it can be formulated as an integer linear programming (ILP) problem. In our solution, we have taken a two-phase approach. The first phase determines an initial coding assignment through *gray code*. In the second phase, a heuristic optimization method based on simulated annealing is applied. Simulated annealing is a well known technique, which is helpful to skip the local minima in the solution space by introducing a probability mechanism [18]. The complete algorithm is outlined in the following pseudo-code.

```
01 OpcodeOptimization($Column\_Graphs$, $Coupling\_Info$)
02 {
03     for each $G_{col}$ in the $Column\_Graphs$ {
04       $G_{col\_undir}$ := MergeEdges($G_{col}$);
05       if $G_{col\_undir}$ contains $\mu$-opcode-nodes {
06         if $G_{col\_undir}$ contains fixed values
07           $SubG_{col\_undir}$ := GetSubgraphWithoutFixedValue($G_{col\_undir}$);
08         else
09           $SubG_{col\_undir}$ := $G_{col\_undir}$;
10         MWP := GetMaximumWeightedPath($SubG_{col\_undir}$);
11         assign $GrayCode_{col}$ values to the nodes in the MWP;
12         calculate $P_{init}$ from $G_{col}$ and $Coupling\_Info$;
13         ReplaceAndSwapWithSA($G_{col}$, $Coupling\_Info$, $P_{init}$);
14       }
15 }}
16
17 ReplaceAndSwapWithSA($G_{col}$, $Coupling\_Info$, $P_{init}$)
18 {
19     $P$ := $P_{init}$;
20     $S(C)$ := get all codings of column bit-width;
21     $T$ := $T_0$, $r$ := factor;
22     while (T $> T_{th}$ ) {
23       for each $\mu$-$op$-$node_i$ in $G_{col}$ {
24         for each $C_i$ in $S(C)$ {
25           $C_{tmp}$ := $C_{\mu-op-node_i}$;
26           assign $\mu$-$op$-$node_i$ with $C_i$;
27           if($C_i$ used by another $\mu$-$op$-$node_j$)
28             assign $\mu$-$op$-$node_j$ with $C_{tmp}$;
29
30           determine $P_{current}$ with new coding of $\mu$-opcode-nodes;
31           diff := $P_{current}$ - $P$;
32           if( (diff<0) or (exp($\frac{-diff}{T}$)>random()) )
33             $P$ := $P_{current}$;
34           else if ($\mu$-$op$-$node_j$ is assigned)
35             UndoSwapCoding($\mu$-$op$-$node_i$, $\mu$-$op$-$node_j$);
36           else
37             UndoAssignment($\mu$-$op$-$node_i$);
38         }
39       }
40       $T$ := $r$*$T$;
41 }}
```

In the first phase of the algorithm, an initial solution is determined. The column graph is at first converted into an undirected graph by merging edges of reverse direction and adding up the edge weights. From the undirected graph, a subgraph is created by removing the nodes with fixed values. To assign binary codes with minimum hamming distance to this undirected graph closely resembles the problem of address assignment during DSP code generation [19]. We adopt the same procedure outlined there. From this subgraph, the Maximum Weighted Hamiltonian Path (MWP) is determined. The gray code is assigned to the nodes of this path. This procedure of gray code allocation is done until all the nodes of the column graph have been assigned to a definite binary coding. An important point of this gray code allocation is to find the ideal starting point in the gray code set. While any continuous chain of gray code elements ensure minimum toggling activity, a particular one ensures minimum transition with the neighbouring columns. Therefore, the starting point of the gray code set is chosen carefully. Note that the removal of the directed edges and the creation of the subgraph is just for preparing the maximum weighted path. The power measurement is performed over the unaltered graph (G_{col}). During the second phase of the algorithm, a heuristic replacement and swapping of μ-opcodes based on simulated annealing algorithm is performed.

For each μ-opcode node, each possible coding in the $S(C)$ is evaluated. If the coding is already assigned to another μ-opcode node, the codings of both μ-opcode nodes are swapped. For each new assignment, the power $P_{current}$ is calculated, considering both toggling and coupling information within the column and coupling information between the column and adjacent columns. If the $P_{current}$ is less than the old power or the difference between them is small enough, as shown in line 32, the μ-opcode nodes keep the newly assigned coding and the old power is updated with $P_{current}$. Otherwise, the coding assignment is undone. After iterating all the μ-opcode nodes, the temperature is reduced for the next annealing step by multiplying it with the factor $r < 1$, until the threshold temperature T_{th} is reached. The random number varies between 0 and 1.

For clarity, in this pseudo-code the whole power consumption is calculated at each temperature for each possible coding of the nodes. In fact, only the difference between the old power and current power needs to be calculated. This is important to reduce the complexity of power calculation and therefore the runtime of the algorithm.

D. RNA Optimization

In comparison to ORA, for the RNA, the complete encoding of a register needs to be considered as a whole. This is due to the fact that a part of a register encoding does not carry a significance like a part of the opcode. Therefore, for the RNA problem, the assembly program and the grammar file are used directly as the entry point. Based on these, toggling information and coupling information for registers are extracted, and correspondingly a graph and a hash table are generated. By applying a heuristic approach to the graph and the hash table, the registers are re-allocated. Note that there is no differece between RNA optimization for RISC architectures and for VLIW architectures.

1) Information Extraction in RNA: From table I and table II, relation between instruction patterns and non-terminals in the grammar file can be derived, which is presented in table X. Based on this table, the toggling information and coupling information between registers, opcodes, immediates and don't cares can be obtained from the generated instruction sequence by simulating the program. Figure 7 illustrates how the information about registers and other items is extracted from an instruction.

TABLE X.
INSTRUCTION PATTERNS AND NON-TERMINALS

insn_0	10rr rrrr rrii iiii	ld reg reg imm
insn_1	0001 rrrr rrrr rrrr	add reg reg reg
insn_2	0010 rrrr rrrr rrrr	sub reg reg reg
insn_3	0011 rrii iiii iiii	jmp cond_reg dst_imm
nop	01xx 0000 0000 xxxx	nop

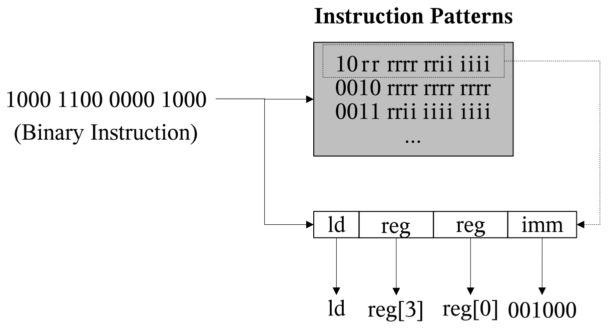


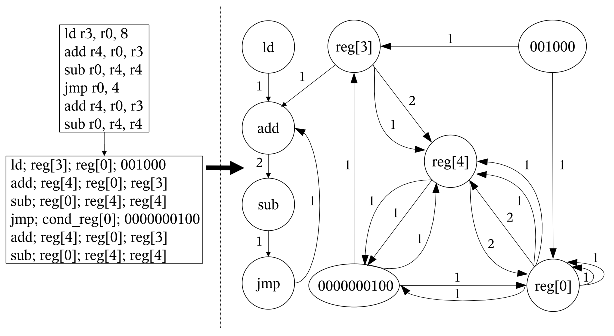
Figure 7. RNA Information Extraction

2) RNA Graph and Hash Table: From the extracted information, a *directed multi-graph* is derived. Formally, the graph is defined as $\langle V, E \rangle$, where V represents an instruction element (register, opcode, immediate etc.) and E is the edge set between two vertices. The edge set contains multiple directed edges between two vertices. Due to the possible spread of a register encoding over different parts of an instruction, it is possible to have an edge, which indicates a partial overlap of the register encoding. As shown in figure 8, the register `reg[3]` has a single directed transition to the opcode of `add`, which is partially overlapped. The details of overlapping bit-width are not shown in the figure for clarity. It is important to note that all register instances with the same index in the assembly program are mapped to the same node. This approach completely removes the complexity of maintaining the define-use chain of registers in the program, while doing the register name adjustment.

Exemplary coupling information between an addition and a subtraction instruction is given in figure 9.

With the graph and the coupling information, the RNA problem can be summarized as the determination of the encoding of the register nodes of the graph, so as to minimize the power consumption contained in the graph and the hash table.

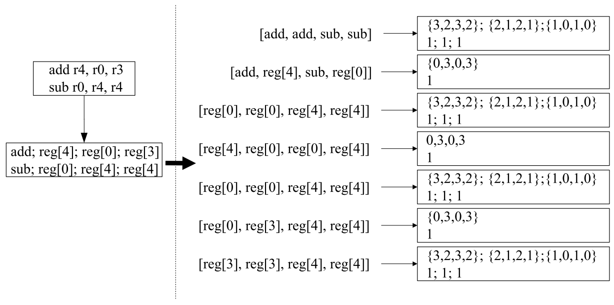
3) Solution of RNA: Since the register nodes in the directed multigraph may have high number of edges



```
ld r3, r0, 8
add r4, r0, r3
sub r0, r4, r4
jmp r0, 4
add r4, r0, r3
sub r0, r4, r4
```

```
ld; reg[3]; reg[0]; 001000
add; reg[4]; reg[0]; reg[3]
sub; reg[0]; reg[4]; reg[4]
jmp; cond_reg[0]; 0000000100
add; reg[4]; reg[0]; reg[3]
sub; reg[0]; reg[4]; reg[4]
```

Figure 8. Directed Multigraph for RNA



```
add r4, r0, r3
sub r0, r4, r4
```

```
add; reg[4]; reg[0]; reg[3]
sub; reg[0]; reg[4]; reg[4]
```

```
[add, add, sub, sub]                 → {3,2,3,2}; {2,1,2,1};{1,0,1,0}
                                       1; 1; 1
[add, reg[4], sub, reg[0]]           → {0,3,0,3}
                                       1
[reg[0], reg[0], reg[4], reg[4]]     → {3,2,3,2}; {2,1,2,1};{1,0,1,0}
                                       1; 1; 1
[reg[4], reg[0], reg[0], reg[4]]     → 0,3,0,3
                                       1
[reg[0], reg[0], reg[4], reg[4]]     → {3,2,3,2}; {2,1,2,1};{1,0,1,0}
                                       1; 1; 1
[reg[0], reg[3], reg[4], reg[4]]     → {0,3,0,3}
                                       1
[reg[3], reg[3], reg[4], reg[4]]     → {3,2,3,2}; {2,1,2,1};{1,0,1,0}
                                       1; 1; 1
```

Figure 9. Coupling Information for RNA

with other instruction elements, it is unlikely to have a continuous chain of register elements. Therefore, the allocation of gray code to the register elements is difficult. We adopted a heuristic approach similar to that performed during ORA, where codings of register nodes are replaced or swapped to improve the overall power consumption. Our solution for the RNA problem is outlined in the following pseudo-code.

```
01 RegisterOptimization($G_{RNA}$, $Coupling\_Info$)
02 {
03       calculate $P_{init}$ from $G_{RNA}$ and $Coupling\_Info$;
04       ReplaceAndSwapWithSA($G_{RNA}$, $Coupling\_Info$, $P_{init}$);
05 }
06
07 ReplaceAndSwapWithSA($G_{RNA}$, $Coupling\_Info$, $P_{init}$)
08 {
09       $P$ := $P_{init}$;
10       $T$ := $T_0$, $r$ := factor;
11       while ($T > T_{th}$ ) {
12         for each $node_{reg_i}$ in $G_{RNA}$ {
13           $S(C_i)$ := GetAvailableCodings($node_{reg_i}$);
14           for each $C_i$ in $S(C_i)$ {
15             $C_{tmp}$ := $C_{reg_i}$;
16             assign $node_{reg_i}$ with $C_i$;
17             if ($C_i$ used by another $node_{reg_j}$) {
18               $S(C_j)$ := GetAvailableCodings($node_{reg_j}$);
19               if ($C_{tmp} \notin S(C_j)$) {
20                 UndoAssignment($node_{reg_i}$);
21                 continue;
22               }
23               else
24                 assign $node_{reg_j}$ with $C_{tmp}$;
25             }
26             determine $P_{current}$ with new coding of registers;
27             diff := $P_{current}$ - $P$;
28             if( (diff<0) or (exp($\frac{-diff}{T}$)>random()) )
29               $P$ := $P_{current}$;
30             else if ($node_{reg_j}$ is assigned)
31               UndoSwapCoding($node_{reg_i}$, $node_{reg_j}$);
32             else
33               UndoAssignment($node_{reg_i}$);
34         }}
35         $T$ := $r * T$;
36 }}
```

The main algorithm of the RNA is controlled by simulated annealing (line 28). For each register node, a set of available codings is reserved. The significance of this set is two-fold. Firstly, it includes the unallocated registers from the processor architecture. Secondly, this allows irregular cases of register access which simple register replacement and swapping is unable to do. For example, a processor may allow addressing of the same register with 2 or 4 bits. With simple replacement and swapping, the 4 bit register encoding will never be exchanged with a 2 bit register, even though it is possible.

While assigning a register with a coding in its available coding list, it is examined in the graph, whether another register node already occupies the coding. If it is the case, the algorithm continues to examine if the second register node can be assigned with the coding of the first register, by checking whether the coding of the first register node exists in the list of available codings of the second register node. Only when the second register node can be assigned with the coding of the first register, a coding swapping is performed between both register nodes. Otherwise, the algorithm removes the assignment for the first register node and continues with its next available coding. After assigning the register nodes, current power with updated coding of the register nodes is calculated, which is compared to the initial power. If the difference between the current power and the initial power fulfills the condition given in line 28, the initial power is updated with current one, otherwise the assignment for the register nodes is changed back.

This presented solution of RNA augments the existing RNA solution [9] by three important steps. Firstly, the storing of the complete transition information in the assembly program in a directed multigraph and a hash table increases the accuracy and efficiency of RNA. Secondly, the heuristic replacement and swapping of register encoding decreases the risk of hitting a local minima in the solution space. Thirdly, the problem of irregular addressing of registers can be easily solved by predefining available codings for the registers.

It is interesting to observe that an approach based on simulated annealing has been successfully employed to reduce the power consumption for buses [20]. It orders the bus lines in the routing phase to minimize the power consumption. The approach presented in this section can be thought of performing a similar re-ordering on a much higher level of abstraction and therefore, complementing the bus re-ordering technique.

V. Optimization Algorithm for Multiple Assembly Programs

The technique for the instruction encoding optimization mentioned in the last section is not limited to one assembly program. It can be extended to the optimization for a group of assembly programs. Typically, an embedded processor can target several applications within or across classes of algorithms. Since all the assembly programs of the applications run on the same processor architecture, the problem of optimization for multiple assembly programs can be formulated as the individual optimization on the software part of the programs, with the opcodes being optimized in common. The optimization flow is outlined in figure 10.

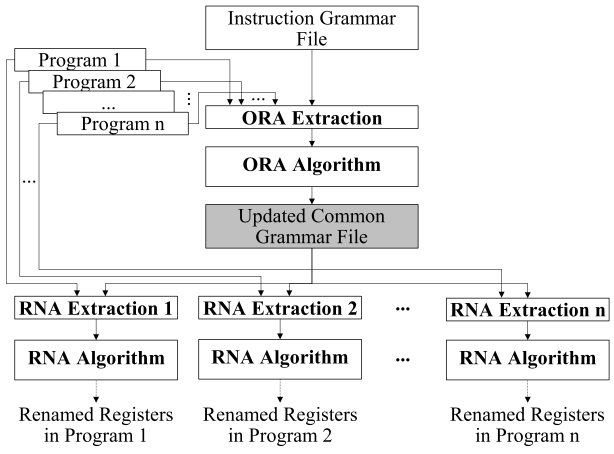


Figure 10. Multiple Assembly Programs Optimization Flow

- Firstly, the columns information is extracted for each assembly program. Out of each program, a set of column graphs and a hash table are created. Based on these, the ORA technique is used aiming at the optimization in all programs.
- After the ORA optimization, the grammar information is updated. The new grammar information is used as the common grammar information for each program at RNA optimization.
- In the last step, RNA optimization is performed for each program individually, based on the common grammar information. Out of each program, a register graph for toggling information and a hash table for coupling information are created for re-assigning the registers.

Since RNA optimization can be applied to different programs individually, which is the same as described in the previous section, only the ORA technique for common opcode optimization for a set of programs is introduced here. With toggling information and coupling information extracted for each program, the ORA algorithm for opcodes optimization for multiple programs is also based on the simulated annealing approach, similar to the ORA algorithm applied to single programs. The main difference between ORA for one single program and ORA for multiple programs is that the condition of simulated annealing for accepting a new coding assigment has to be met for all programs. A brief pseudo-code of the ORA optimization for multiple programs is shown in the following.

```
01 ReplaceAndSwapForMultiAppWithSA($G_{col}\_List$, $Coupling\_Info\_List$)
02 {
03       for each $program_i$ of all programs {
04         $P_{pg\_i_{init}}$ := get initial power of $program_i$;
05       }
06       $T$ := $T_0$, $r$ := factor;
07       while($T > T_{th}$) {
08         $S(C)$ := get all codings of column bit-width;
09         $List_{\mu-op-nodes}$ := get all $\mu$-opcodes from $G_{col}\_List$;
10         for each $\mu$-$op$-$node_i$ in $List_{\mu-op-nodes}$ {
11           for each $C_i$ in $S(C)$ {
12             $C_{tmp}$ := $C_{\mu-op-node_i}$;
13             assign $\mu$-$op$-$node_i$ with $C_i$;
14             if($C_i$ used by another $\mu$-$op$-$node_j$)
15               assign $\mu$-$op$-$node_j$ with $C_{tmp}$;
16
17             improve_all := true;
18             random_number := random();
19             for each $program_i$ of all programs {
20               determine $P_{pg\_i_{current}}$ with new codings of $\mu$-opcode-nodes;
21               diff := $P_{pg\_i_{current}}$ - $P_{pg\_i_{init}}$;
22               if( (diff>0) and (exp($\frac{-diff}{T}$)<random_number) ) {
23                 improve_all := false;
24                 break;
25             }}
26
27             if (improve_all == true)
28               for each $program_i$ of all programs
29                 $P_{pg\_i_{init}}$ := $P_{pg\_i_{current}}$;
30             else if ($\mu$-$op$-$node_j$ is assigned)
31               UndoSwapCoding($\mu$-$op$-$node_i$, $\mu$-$op$-$node_j$);
32             else
33               UndoAssignment($\mu$-$op$-$node_i$);
34           }}
35           $T$ := $r*T$;
36 }}
```

The random mechanism is shown in line 22. For each program, its current power is compared with the old power. Only when the result of the comparison satisfies the requirement of the simulated annealing process for each program, the condition for accepting the new coding assignment is met, otherwise, the new assignment is changed back. Here, as seen in line 18, a same random number is used for all programs at each coding assignment iteration.

Since the stochastic information about the frequency of running each program is absent, we assume that the probability of running each program is the same. Hence, in this pseudo code the condition of simulated annealing has to be satisfied for each program (line 17-25). If the stochastic information is known, the power consumption on each program can be weighted. By summing up the weighted power consumption on each program, an overall power consumption can be calculated. Then this overall power consumption can be used to determine whether a new coding assignment is to be taken or not.

VI. Results

The optimizations discussed in this paper are tested with four different processors. The **ICORE** [7] architecture is dedicated for Terrestrial Digital Video Broadcast (DVB-T) decoding. We took two assembly programs *cordic01* and *cordic02* running on this architecture. Both assembly programs perform the same cordic algorithm, but with different implementations. The second architecture of our case study is **LEON3**. LEON3 [21] is a 7-stage pipelined processor, compliant with the SPARC V8 architecture. Two programs running on LEON3 namely, *integer matrix multiplication* (mmul), and *bubblesort* are used for the case study. The third processor is for real-time **Retinex** image and video filtering [22], which is also a 7-stage pipelined architecture. Two image processing programs *color treatment* (col_treat) and *image enhancement* (img_enhance) are run on the architecture. The last processor architecture that we studied is **TriMedia DSPCPU32**, which is based on a 5-slot VLIW architecture with 6 pipeline stages. On this architecture we ran two programs, namely, *Adaptive Differential Pulse Code Modulation* (adpcm) and *blowfish encryption-decryption* (blowfish). The value of λ used for the measurements in table XII, table XIII and table XIV is 3.

All the abovementioned architectures are implemented using an ADL description. The instruction-set grammar is automatically extracted from the ADL description. Note that a hand-written instruction-set grammar can also be used to employ the optimizations outlined. The optimized assignment for the opcode is updated in the ADL and the new HDL implementation is automatically generated from that. The register name adjustment is performed directly on the assembly program. The modified program is run on the generated HDL description of the processor to check the correctness of the optimized encoding. Finally, the original and optimized toggling activity as well as the coupling transitions are measured.

A. Power Reduction in the Instruction Memory

Table XI summarizes the toggling activity reduction in the instruction word lines.

TABLE XI.
TOGGLING ACTIVITY REDUCTION

architecture	*program*	*reduction*
ICORE	cordic01	50.00%
	cordic02	20.32%
LEON3	mmul	24.22%
	bubblesort	23.53%
Retinex	col_treat	36.11%
	img_enhance	34.46%
TriMedia	adpcm	24.12%
	blowfi sh	19.69%

B. Memory Power Reduction : System-level Effect

In order to investigate the system-level effect of these savings, the power consumed by an ICORE processor and memory are measured using commercial RTL power measurement tools [23]. In a typical use-case scenario, 32.4% of the total power is consumed by the instruction memory. The toggling activity improvement corresponds to a strong instruction memory power savings, assuming 60% of the memory power is consumed due to this toggling activity [11]. For ICORE, this corresponds from **3.95%** up to **9.72%** of the overall power consumption. Although the overall effect is not dramatic, yet it is significant as it is achieved without any performance or area overhead.

C. Power Reduction in the Instruction Bus

The power reduction in the instruction bus strongly depends on the value of capacitances, bus-length, bus-width, supply voltage and other parameters. The absence of an RTL power measurement tool with deep submicron bus power model prevented us from obtaining precise bus power consumption as part of the overall system. In existing literature, often a specific value of abovementioned parameters is chosen to present the power reduction [14]. In this paper, in order to maintain a fair basis for comparison, a model from equation 3 is extracted without the capacitance, frequency and supply voltage. This is referred as Power Consumption Cost (PCC). Table XII summarizes the improvement of PCC for the four target processors.

$$PCC = (\alpha + \beta\lambda + 2\gamma\lambda) \tag{3}$$

TABLE XII.
OVERALL PCC IMPROVEMENT

architecture	*program*	*original*	*optimized(improvement)*
ICORE	cordic01	4310	2237 (48.10%)
	cordic02	1603	1223 (23.71%)
LEON3	mmul	58660	43150 (26.44%)
	bubblesort	112045	79989 (28.61%)
Retinex	col_treat	756131	437598 (42.13%)
	img_enhance	266669	168526 (36.80%)
TriMedia	adpcm	6903677	4808683(30.35%)
	blowfi sh	15869089	11792815(25.69%)

Note that the instruction-set for the ICORE architecture was already optimized for the two test programs using the techniques outlined in [7]. Their approach did not consider the coupling effects. The optimizations by the RNA technique (not covered in [7]) also contributed to the strong reduction of power consumption.

Figure 11 shows the overall improvement for the programs with λ varying from 0 to 4. It can be observed that the improvement of PCC approaches a saturation value, for $\lambda \rightarrow \infty$. We can also observe that the simulated annealing algorithm introduces slight variations due to the random mechanism.

D. Separate Optimization Contributions

The individual contributions in power consumption improvement by the ORA and the RNA technique are shown in table XIII. For both the ORA and the RNA, the improvements are measured across the complete instruction, not only for the modified parts of the coding. The improvements, therefore, are indices of the individual width and organization of the coding elements in the total instruction. The main reason, why the improvement based on ORA optimization for LEON3 and TriMedia is relatively low, is that the opcode is arranged sparsely in the instruction word compared to the regular opcode organization in ICORE and Retinex. The low improvement

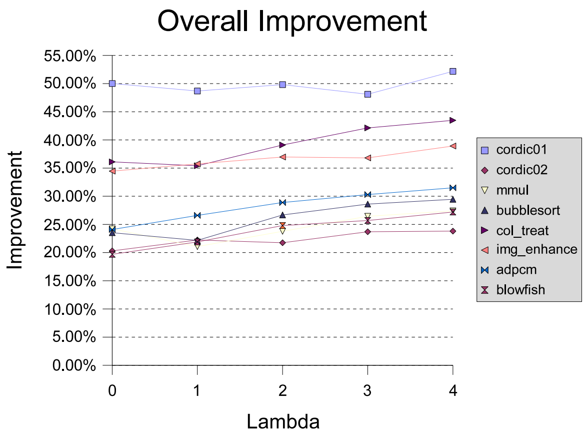


Figure 11. Overall Improvement with Variable λ

based on RNA for the program *cordic02* is caused by a customized instruction in the architecture, which uses 6 out of 8 general purpose registers as dedicated registers for a special functionality. So those registers must not be changed in this program. Hence, the scope of the optimization in the register field of the program is very limited. In contrast, since this special instruction is not used in the assembly program *cordic01*, the limitation of the scope of the optimization does not exist. At the design phase of the processor architecture, this limit can still be taken off by modifying the architecture and the program correspondingly together. In this case for the optimization on *cordic02*, the improvement of PCC can reach more than 11%, by changing both the architecture and the assembly program.

TABLE XIII.
SEPARATE PCC IMPROVEMENT BY ORA AND RNA

architecture	*program*	*ORA*	*RNA*
ICORE	cordic01	24.66%	17.08%
	cordic02	22.46%	1.81%
LEON3	mmul	4.28%	20.80%
	bubblesort	6.89%	22.39%
Retinex	col_treat	23.61%	17.13%
	img_enhance	24.27%	16.63%
TriMedia	adpcm	7.69%	23.35%
	blowfi sh	9.89%	15.56%

Note that the sum of improvement by ORA and by RNA is not necessarily the same as the improvement by RNA after ORA in table XII. The reason is that in the latter case after the ORA optimization, the coding characteristic in the programs is changed. Based on the changed program, the following RNA may produce different result than the RNA based on the original program.

E. Comparison with Greedy Approach for RNA

In order to show the efficacy of our approach, we have compared the approach based on simulated annealing with the greedy approach for RNA outlined in [9]. The results of these comparisons for the four processors are given in table XIV. In all the cases, the simulated annealing is performing better compared to the greedy one. In some cases, the greedy algorithm is actually producing worse PCC results than original PCCs. The reason is that the greedy algorithm outlined in [9] does not take coupling effect into account.

TABLE XIV.
COMPARISON WITH GREEDY APPROACH

architecture	*program*	*Improvement*	
		Greedy	*SA*
ICORE	cordic01	-8.65%	17.08%
	cordic02	1.56%	1.81%
LEON3	mmul	7.39%	20.80%
	bubblesort	0.34%	22.39%
Retinex	col_treat	6.48%	17.13%
	img_enhance	-3.89%	16.63%
TriMedia	adpcm	-10.2%	23.35%
	blowfi sh	-8.91%	15.56%

Furthermore, the greedy approach does not have the ability to know the information about the global directions of the edges in the graph. Therefore, such an approach is inherently limited in its ability. The following example shows the limitation of greedy method clearly. In figure 12, a simple register graph is shown. The initial indices for r0, r1, r3 are “00”, “01” and “11” respectively. For simplicity, here we don’t consider the coupling effect. So the initial PCC is zero. According to the greedy method, the registers r0 and r1 are assigned at first, because the weight of the edge between them are the largest in the graph. Assuming the registers r1 and r0 are assigned with “11”and “10”, which surely makes the toggling from r1 to r0 to zero, the best index left for r3 is “01”. It results in a PCC of 3. In comparison to the initial zero-PCC, the result gets worse. So for already quite optimized encoding, using the greedy method has potential risk, getting a worse result.

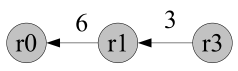


Figure 12. Example for Optimization with Greedy Method

F. Comparison with Odd/Even Bus Invert Encoding Technique

It is worthwhile to compare the power savings achieved in the proposed framework to those achievable by pure encoding-decoding techniques. We select one such technique namely, Odd/Even Bus Invert Encoding Technique (OEB) [15], which considers the effect of coupling capacitance and adds two extra bus-lines. Depending on the code transmitted in these extra lines, the data is encoded and decoded by inverting odd or even lines of the original bus. As can be observed from figure 13, the instruction encoding techniques presented here outperforms the OEB technique. In some cases, where the bus lines are already in low consumption state, the OEB technique even gets worse result due to the added power consumption of extra bit lines.

G. Bus Power Reduction : System-level Effect

In order to judge the effect of the above reductions on the overall system, the scaling trends of interconnect

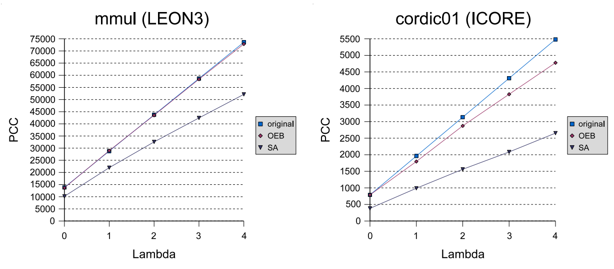


Figure 13. Comparison with OEB

power with technology are considered. Based on [24], the following results were obtained. For microprocessor-based systems, on-chip signaling interconnects are reported to constitute 46% of overall power in 180 nm technology and predicted to contribute 27% of overall power in 50 nm technology. As indicated in the results, our proposed techniques save from **25.69%** up to **48.10%** of the instruction bus power, which is 11.82% to 22.13% of total power in 180 nm technology and 6.93% to 12.99% in 50 nm technology.

H. Runtime

The simulated annealing is a computation-intensive heuristic. In our case, the algorithm runtime largely depends on the complexity of the instruction grammar. Note that the size of the program affects the run time only if the additional instructions create a new graph edge or a new coupling information. Therefore, beyond a certain size the run time is affected by the programs only slightly.

We ran the experiments on an AMD Athlon(tm) 64x2 processor mit 2.4GHz CPU and 2G Memory. The whole run time including ORA and RNA for ICORE, LEON3 and Retinex is under 10 Minutes. For Trimedia, the run time is around 1 hour. The main reason is that the registers in this architecture are encoded with 7 bit and a huge number of registers are used in the programs. The number of iterations on the registers and the available codings for registers is increased significantly, in comparison to other architectures.

VII. Summary

The decreasing power budget of modern application-specific processors have created strong interest towards low-power design techniques. Instruction memory and bus, being two strong power consuming components of a system, are of special interest in a low power processor design methodology. Hence, an efficient instruction-set encoding is required for reducing power consumption in the instruction bus and the instruction memory. In this paper, this problem is addressed. In the following, the limitations and contributions of this paper are summarized.

Limitation: The limitation of the approach presented in this paper can be readily appreciated by comparing with the work presented by Petrov et al [10]. In our approach, the instruction encoding is determined during processor design phase and therefore, one instruction can have one unique opcode. In the approach outlined by Petrov et al, the program binary is transformed considering purely the bit-sequences. This technique (and likewise other sophisticated bus encoding techniques) can potentially have more flexible transformations.

From another point of view, the approach of [10] is a processor-independent encoding technique and thus needs additional circuitry for encoding and decoding. In contrary, our approach is strongly dependent on the organization of the processor instructions. The ORA approach presented in this paper is highly beneficial for application-specific processor design, where the program set is well known in advance. The power savings under that scenario is significant and is achieved without any associated overhead. Understandably, our approach can be combined with other processor-independent approaches ([10], [12]) to reap further benefits.

Contribution: The contributions of this paper are twofold. Firstly, this paper contributes a framework and technique for determining low-power instruction opcode (ORA) during application-specific processor design. Secondly, this paper contributes a novel algorithm for register name adjustment (RNA), which shows better results than the existing greedy algorithm.

Outlook: In the future, our work will focus on the combination of the ORA and RNA techniques with available bus encoding techniques to optimize the power consumption. It will also be highest interesting to integrate the RNA technique into compiler to get power efficient allocation of registers, with additional information provided by the profiling of the programs.

References

[1] P. Grun, A. Halambi, A. Khare, V. Ganesh, N. Dutt, and A. Nicolau, "EXPRESSION: An ADL for System Level Design Exploration," Department of Information and Computer Science, University of California, Irvine, Tech. Rep., 1998.

[2] A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002.

[3] *http://www.retarget.com*, Target Compiler Technologies.

[4] *http://www.tensilica.com*, Tensilica.

[5] *http://www.stretchinc.com*, Stretch.

[6] T. Mudge, "Power: A First-Class Architectural Design Constraint," *Computer*, 2001.

[7] T. Gloekler and H. Meyr, *Design of Energy-Effi cient Application-Specifi c Instruction Set Processors*. Springer, 2004.

[8] L. Benini, G. De Micheli, A. Macii, E. Macii and M. Poncino, "Reducing Power Consumption of Dedicated Processors through Instruction Set Encoding," 1998.

[9] P. Petrov and A. Orailoglu, "Transforming Binary Code for Low-Power Embedded Processors," *IEEE Micro*, vol. 24, no. 3, 2004.

[10] P. Petrov and A.Orailoglu, "Low-power instruction bus encoding for embedded processors," *IEEE Trans. Very Large Scale Integr. Syst.*, 2004.

[11] M. F. Chang and M. J. Irwin, and R. M. Owens, "Power-Area Tradeoff in Divided Word Line Memory Arrays," *Journal of Circuits, Systems, Computers*, vol. 7, no. 1, 1997.

[12] E. Macii, M. Poncino and S. Salerno, "Combining Wire Swapping and Spacing for Low-power Deep-submicron Buses," in *GLSVLSI '03: Proceedings of the 13th ACM Great Lakes symposium on VLSI*. New York, NY, USA: ACM Press, 2003.

[13] P. P. Sotiriadis and A. Chandrakasan, "Bus Energy Minimization by Transition Pattern Coding (TPC) in Deep Submicron Technologies," in *ICCAD '00: Proceedings of the 2000 IEEE/ACM International Conference on Computer-aided Design*. IEEE Press, 2000.

[14] L. Macchiarulo, E. Macii and M. Poncino, "Low-energy Encoding for Deep-submicron Address Buses," in *ISLPED '01: Proceedings of the 2001 International Symposium on Low Power Electronics and Design*. ACM Press, 2001.

[15] Y. Zhang and J. Lach and K. Skadron and M. R. Stan, "Odd/even Bus Invert with Two-phase Transfer for Buses with Coupling," in *ISLPED '02: Proceedings of the 2002 International Symposium on Low power Electronics and Design*. ACM Press, 2002.

[16] A. C. Cheng and G. S. Tyson, "An Energy Efficient Instruction Set Synthesis Framework for Low Power Embedded System Designs," *IEEE Trans. Comput.*, vol. 54, no. 6, 2005.

[17] A. Nohl and G. Braun and O. Schliebusch and R. Leupers, H. Meyr and A. Hoffmann, "A Universal Technique for Fast and Flexible Instruction-set Architecture Simulation," in *DAC '02: Proceedings of the 39th conference on Design automation*. ACM Press, 2002.

[18] S. Kirkpatrick, C. D. Gelatt and M. P. Vecchi, "Optimization by Simulated Annealing," *Science, Number 4598, 13 May 1983*, vol. 220, 4598, pp. 671–680, 1983.

[19] R. Leupers and P. Marwedel, "Algorithms for Address Assignment in DSP Code Generation," in *ICCAD '96: Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society, 1996.

[20] Y. Shin and T. Sakurai, "Coupling-driven Bus Design for Low-power Application-specific Systems," in *DAC '01: Proceedings of the 38th conference on Design automation*. New York, NY, USA: ACM Press, 2001.

[21] *http://www.gaisler.com/*, Gaisler Research.

[22] S. Saponara, L. Fanucci, S.Marsi, G. Ramponi, D. Kammler, E. M. Witte, "Application-specific instruction-set processor for retinex-like image and video processing," *IEEE Transactions on Circuits and Systems - II: Express Briefs*, vol. 54, no. 7, July 2007.

[23] *Prime Power http://www.synopsys.com/products/power/primepower_ds.pdf*, Synopsys.

[24] G. Chandra, P. Kapur and K.C. Saraswat, "Scaling Trends for the On Chip Power Dissipation," *IEEE Interconnect Technology Conference*, 2002.

Diandian Zhang received the Diploma degree in electrical engineering from the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany, in 2006. He is currently a Ph.D. candidate in electrical engineering at RWTH Aachen University. His current research focuses on architecture exploration and implementation for application specific processors and multiprocessor System-on-Chips (MPSoCs).

Anupam Chattopadhyay received the Master of engineering in embedded systems design from the University of Lugano, Switzerland, in 2002 and is currently pursuing the Ph.D. degree from the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany. His research interests include automatic implementation of processors with LISA, architecture optimization techniques and a tool flow for reconfigurable ASIPs.

David Kammler received the Diploma degree in electrical engineering from the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany, in 2003 where he is currently pursuing the Ph.D. degree. His research interests include ADL based automatic implementation of processors, architecture exploration and implementation and fault tolerant processor design.

Ernst Martin Witte received the Diploma degree in electrical engineering in 2004 from the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany, where he is currently pursuing the Ph.D. degree. His research focuses on architecture exploration and implementation with special respect to application specific processors and the field of Software Defined Radio (SDR).

Gerd Ascheid is professor and director of the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany. He holds a Diploma and a Ph.D. degree in electrical engineering (RWTH Aachen University, 1977 and 1983). He was a co-founder of CADIS GmbH (acquired by Synopsys in 1994) and later a senior Director at Synopsys. His research interests include advanced modems for wireless communication and their efficient implementation using digital signal processing.

Rainer Leupers received the Diploma and Ph.D. degrees in Computer Science with honors from the University of Dortmund, Germany, in 1992 and 1997. Since 2002, he has been professor for Software for Systems on Silicon (SSS), RWTH Aachen University, Germany. He was a co-founder of LISATek Inc. (acquired by CoWare Inc. in 2003). His research activities revolve around software development tools, processor architectures, and electronic design automation for embedded systems, with emphasis on C compilers for application specific processors in the areas of signal processing and networking.

Heinrich Meyr received his M.Sc. and Ph.D. from ETH Zurich, Switzerland. From 1977 to 2007, he was professor and director of the Institute for Integrated Signal Processing Systems (ISS), RWTH Aachen University, Germany. He was a co-founder of CADIS GmbH (acquired by Synopsys in 1994) and a co-founder of LISATek Inc. (acquired by CoWare Inc. in 2003). As well as being a Fellow of the IEEE he has served as Vice President for International Affairs of the IEEE Communications Society. His research interests include the interactive design of receiver algorithms for advanced wireless systems and their implementation as heterogenous platforms.