CHAPTER 4 DECOMPOSITION OF INSTRUCTION DECODER FOR LOW POWER DESIGNS

Most previous studies [18-26] focused on the power reduction of fetching instructions. Only a few studies focused on power reduction for instruction decoding and control signal generation. A two sub-decoders structure [27] was proposed for the power minimization on instruction decoders. In [28], two types of instruction decoders were designed for Pentium Pro: one for simple instructions and the other for complex instructions. This structure was proposed for performance improvement rather than power minimization. The studies have also shown that the instruction decoder can consume as much as 18% of the system power in *StrongArm* [4] and 14% in Pentium Pro [24]. Thus, this paper focuses on the reduction of instruction decoder power usage.

In this chapter, we present two techniques for decomposing instruction decoders to minimize power consumption. First, based on the irregular execution frequency of instructions, we propose to decompose an instruction decoder into two or more coupling sub-decoders, a process called horizontal decomposition. With this approach, only one small sub-decoder is activated at a time. Second, based on a pipelined decoder structure, we partition the decoder into two pipelined stages, a process called vertical decomposition. The first stage identifies instruction types, while the second stage generates control signals. With this approach, only the second stage is activated in the subsequent execution stages.

4.1. Horizontal decomposition

The instruction-set of a processor consists of many types of instructions, including arithmetic/logic, data transfer, decision-making, sequencing, and operations that handle exceptions. The execution occurrences of instructions in various application programs may vary in a wide range [43]. For example, exception-handling instructions are executed much less frequently compared to arithmetic/logic-instructions. Moreover, for each specific type of application, the programs exhibit similar behaviors and instruction patterns. For example, in most signal-processing programs, large percentages of the execution time are consumed in loops. Hence, the instructions within loops have higher execution frequencies than those outside of the loops.

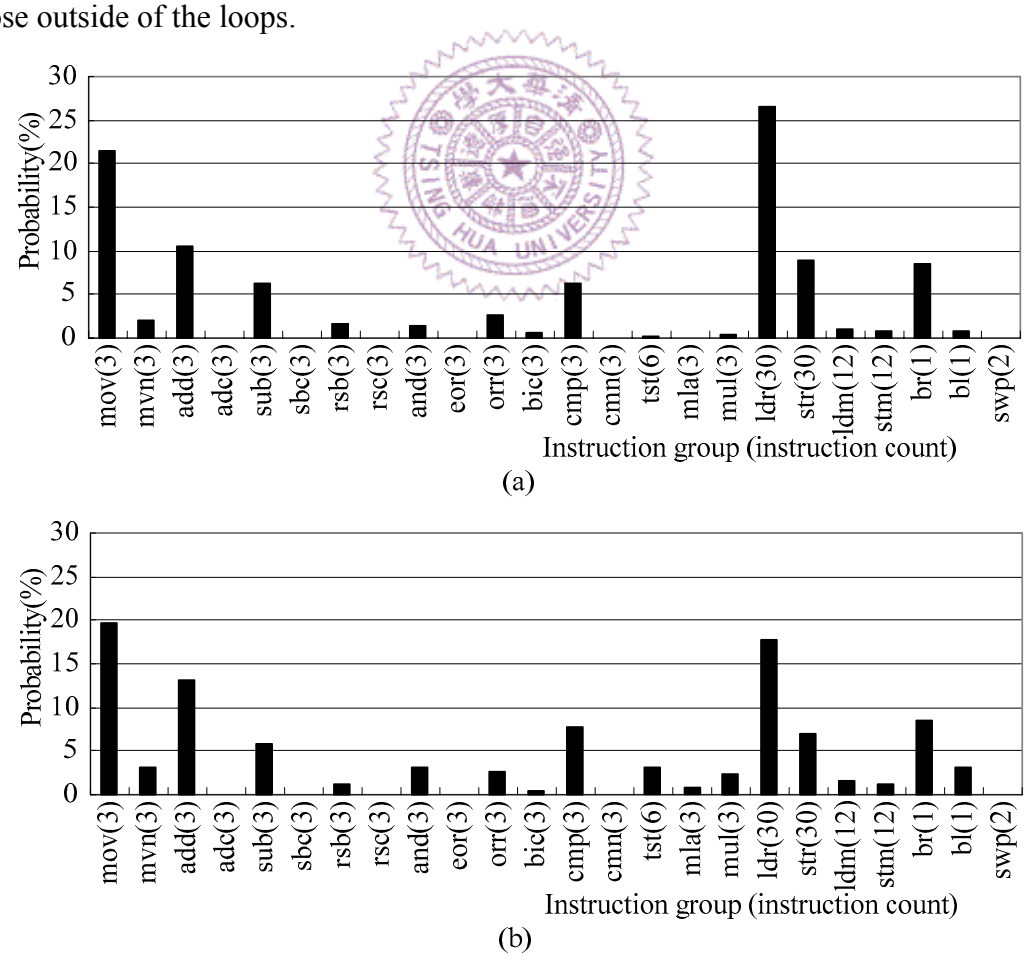

Figure 4.1: The instruction execution-frequency examples of (a) the *Powerstone* and (b) the *MediaBench*.

Figure 4.1 illustrates the profiling of the instruction execution-frequency by executing the benchmark set of Motorola's *Powerstone* [44], which consists of a set of benchmark programs targeted to various applications, including compression, quicksort, and automotive control codes. The set is compiled and profiled on the emulator of *SimIt ARM v1.0* [45]. Figure 4.1(a) shows that the three instructions in the *MOV* class are executed very frequently, with a 22% probability. On the other hand, some instructions are never executed. Figure 4.1(b) shows the profiling result of another benchmark set: the *MediaBench*. Although the instruction distributions are different from those distributions of *Powerstone*, the characterization of instruction execution frequency is the same. That is, the *MOV* and *LDR/STR* categories are the most frequently executed instruction types.

These observations indicate that active instructions occur only within a sub-set of all instructions. Intuitively, instruction-decoding circuit can be decomposed into two sub-circuits, one for the three *MOV* instructions and the other for the rest of the instructions. In this case, one instruction-decoding sub-circuit can be turned off while executing the other one and hence reduce the overall power dissipation. This motivates us to develop a decomposed architecture for low-power decoders.

4.1.1 The decomposition architecture

As discussed in the previous section, active instructions occur only within a subset of all instructions. Based on this observation, we propose a decomposed-decoding method [46] to decompose an instruction decoder into a number of coupling sub-decoders such that only a small decoder will be activated at any time. Figure 4.2 shows that the decoder is decomposed into two coupled sub-decoders, *Sub-Decoder*$_0$ and *Sub-Decoder*$_1$.

The control logic to turn on/off sub-decoders consists of three parts: *Activate-Control*, *input AND-ORs*, and *output ORs*. The operation of the decomposed *Control Logic* is described as follows. *Activate-Control* determines which sub-decoder will be activated by decoding the input instruction. There are two output signals, $Control_0$ and $Control_1$. When $Control_0 = 0$, $Sub\text{-}Decoder_0$ is on and $Sub\text{-}Decoder_1$ is off. In contrast when $Control_1 = 0$, $Sub\text{-}Decoder_1$ is on and $Sub\text{-}Decoder_0$ is off. Turning off $Sub\text{-}Decoder_0$ ($Sub\text{-}Decoder_1$) is controlled by the *AND* and *OR* gates in front of the sub-circuits. When the control signal $Control_0$ ($Control_1$) is 1, it inhibits the propagation of other inputs to an *OR* gate and the inverted value inhibits inputs to an *AND* gate. The outputs of all *OR* gates and *AND* gates will be 1s and 0s, respectively. The vector formed by these 1s and 0s is called the *turning-off vector*. If the sub-circuit remains off for the subsequent instruction, the *turning-off vector* will remain the same and there will be no signal transition and hence, no power consumption. As for the outputs of *Sub-Decoders*, the output OR gates are used to select the outputs of the activated sub-circuit. When $Sub\text{-}Decoder_0$ ($Sub\text{-}Decoder_1$) is off, the output is set to all 0s for the *turning-off vector*. Since 0 is a non-controlling value for *OR* gates, the outputs are determined by $Sub\text{-}Decoder_1$ ($Sub\text{-}Decoder_0$).

Note that in using this decomposition architecture, when one sub-decoder is turned off and relinquishes the control to the other sub-decoder, both sub-decoders will be activated and hence consume power for one instruction execution: The sub-decoder that relinquishes the control will set its inputs to 0 in one clock cycle and in the next clock cycle, the inputs remain 0 to prevent the circuit from transiting in order to reduce dynamic power consumption.

Next, the *AND-OR* gates (*turning-off vector*) is constructed in front of the *Sub-Decoders* to inhibit the propagation of inputs by selecting a *minterm* that is *don't care* to the sub-circuit. For example, for a four-input sub-circuit with *1101* as *don't*

care, we construct *OR-OR-AND-OR* gates in front of the sub-circuit. For this input combination of *1101*(*turning-off vector*), we assign all 0s to the output of the sub-decoder. Since the decoder is decomposed into more than one sub-decoder, the *care minterm* of other sub-decoder is *don't care* for this sub-decoder. Thus, such a '*don't care' minterm* can always be found.

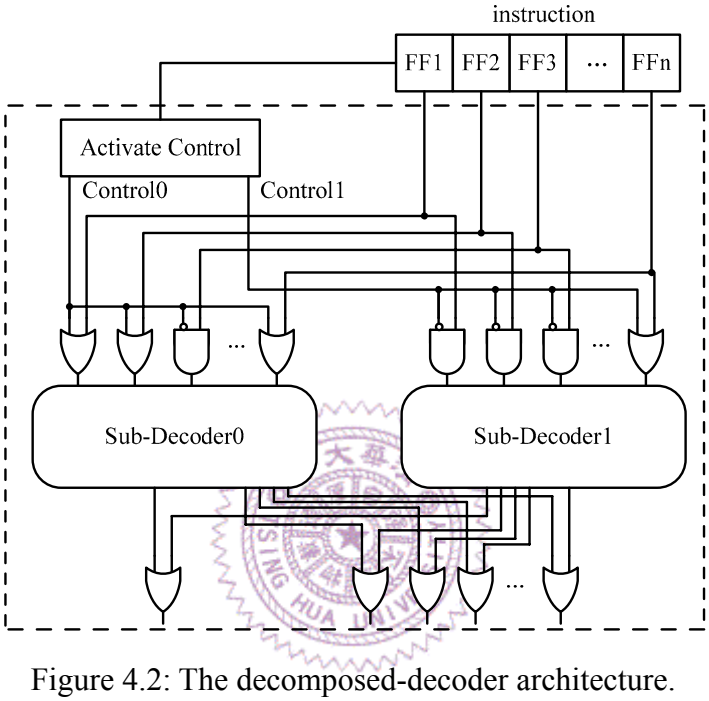

Figure 4.2: The decomposed-decoder architecture.

The issues remaining to be solved are (1) how to partition instructions so that the instruction decoding has a high frequency of execution in a small sub-circuit and the transition among sub-decoders is infrequent and (2) how to take the overhead of *Activate Control* for the instruction decoder (decoder that determines the active *Sub-Decoders*) into consideration when performing the decomposition. We explain how to solve the two issues in the next sub-section.

4.1.2 The decomposition algorithm

The behavior of an application-specific system design, which is unchanging, is usually known. Hence, it is feasible to analyze the application programs and utilize

their characteristics to further optimize the systems. First, we collect a set of instruction-execution sequences from various application programs. Then, we model the execution of instructions as a graph and transform the instruction decomposition into a graph-partitioning problem. The goal is to find a partitioned graph solution given a graph and the number of partitions such that a pre-defined power cost function is minimized.

To apply the graph partition algorithm, we model the instruction-execution sequences as a weighted graph *G = (V, WV, E, WE)*, where *V* is a node set, *WV* the weights on the nodes, *E* an edge set, and *WE* the weights on the edges. A node *iv* in *V* represents one instruction *IV* of the instruction set, the weight on the node *iv* represents the execution frequency of the instruction *IV*, an edge between node *ia* and node *ib* represents that instruction *IA* is executed before instruction *IB* or instruction *IB* is executed before instruction *IA*, respectively, and the weight on the edge represents the transition frequency between *IA* and *IB*.

Figure 4.3 shows an example of constructing the weighted graph from an execution sequence. In Figure 4.3(a), there are five types of instructions in the program sequence. Hence, there are five nodes in the graph as shown in Figure 4.3 (b). The numbers next to the nodes are the execution frequencies of the instructions. For example, *mov* is executed 5 times, as marked by dots in Figure 4.3(a), and there are 15 instructions in total. As a result, the execution frequency of node *mov* is 5/15. The edges between nodes and the weights on the edges are constructed by tracing the instruction execution sequence. For instance, in Figure 4.3(a), the three arrows show that the *mov* is executed before *mul* twice, *mul* is executed before *mov* once, and the total number of transitions is 14. Hence, there is an edge between nodes *mov* and *mul* and the weight of the edge is 3/14.

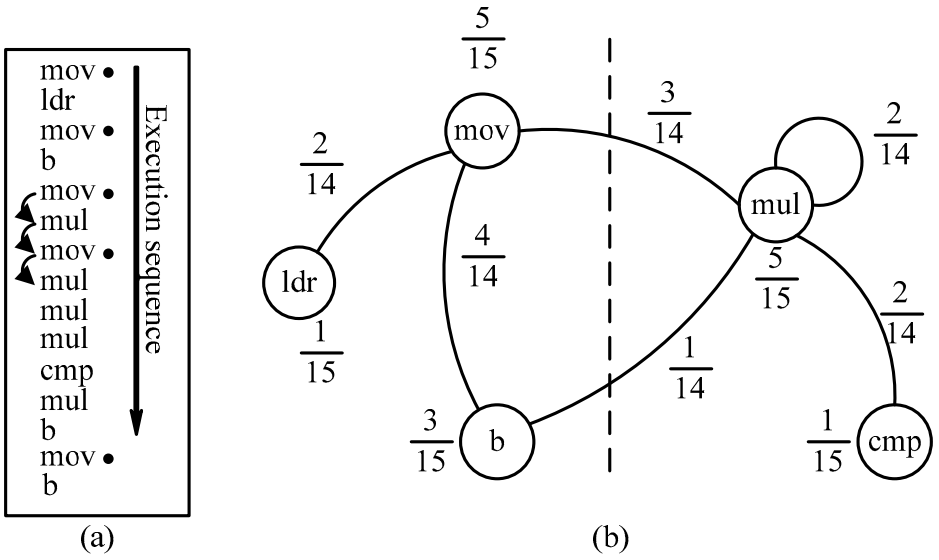

Figure 4.3: A transition graph example.

After constructing an instruction-execution graph, we apply a partitioning algorithm to partition the graph. The cost function for our partitioning algorithm is based on the power estimation between partitioned groups, which is defined as follows. Suppose the original instruction-execution graph G containing S instructions is partitioned into $\pi = (G_1, G_2, ..., G_n)$ sub-graphs where sub-graphs G_i has S_i instructions. Let SF_i denotes the transition frequency of instructions in G_i transiting to G_i itself, and CF_{ij} the frequency of transition between instructions in G_i and instructions in G_j (where $i \neq j$). Hence, SF_i is calculated as the sum of transition frequencies (weights) of edges that connect two instructions in G_i, and CF_{ij} is calculated as the summation of transition frequencies of edges that cross two sub-graphs, G_i and G_j. For example, let the transition graph shown in Figure 4.3(b) be partitioned into two sub-decoders: *Sub-Decoder*$_0$ and *Sub-Decoder*$_1$ and *Sub-Decoder*$_0$ decodes instructions, *mov*, *ldr*, and *b* and *Sub-Decoder*$_1$ decodes instructions, *mul* and *cmp*. Then, SF_1 is 6/14, SF_2 4/14, and CF_{12} 4/14.

Next, power estimation of a sub-graph G_i, $power_i$, is estimated from the result of *report_power* after synthesizing the sub-graph with the *Synopsys Design Compiler* [47]. Finally, the total power consumption is estimated as:

$$\textbf{Power\_Cost}(G,\pi) = \sum_{i=1}^{n} SF_i \times power_i + \sum_{i,j} CF_{ij} \times (power_i + power_j) + overhead(\pi).$$

The first term represents the estimated power consumption for each sub-graph. The second term represents the estimated power consumed during cross transition. When cross transition happens, two involved sub-graphs are activated and hence both consume power. The third term, *overhead(π)*, is the power overhead of activate logic and input/output control logic. It is also estimated by the *report_power*. Take the circuit shown in Figure 4.2 as an example. The overhead includes activate control (it is a one-to-two decoder if the number of sub-decoders is 2), the AND/OR gates in front of *Sub-Decoder*$_0$ and *Sub-Decoder*$_1$, and the OR gates at the outputs of *Sub-Decoder*$_0$ and *Sub-Decoder*$_1$.

In our implementation, we use a three-step algorithm [46] to partition the transition graph into n groups, where n is the desired partition number. First, the algorithm clusters graph nodes into c clusters by a random walk procedure. Then, the $n-1$ clusters with the largest steady instruction frequencies are selected as the seeds of $n-1$ groups, $G_1, G_2, \ldots, G_{n-1}$. The remaining clusters, $C_n, C_{n+1}, \ldots, C_c$, form the last group, G_n. Third, a cluster in G_n is iteratively selected for a possible movement into one of the other groups $G_1, G_2, \ldots, G_{n-1}$, according to the estimated power consumption based on **Power_Cost***(G,π)*. In each iteration, a cluster C_k in G_n is selected. Power consumption is evaluated for all the possible movements of C_k into one of $G_1, G_2, \ldots, G_{n-1}$, as well as for the possibility of leaving C_k in G_n. The move with the lowest power cost will be selected. The reason behind this heuristic is that when a cluster is moved into a group, it increases the steady instruction frequency of the group and changes the transition frequency of the related groups. Partition results with high steady-instruction frequencies, low transition frequencies, and low power are desirable.

4.2. Vertical decomposition

The instruction decoder for pipelined control-signal generation also affects the power consumption of processors. In this section, we propose a vertical decomposition method to reduce power consumption. First, we review two basic pipelined decode structures. Then, we present our decomposed decoding structure for low power designs. Finally, we present an algorithm for intermediate code assignment to reduce the area cost incurred by the proposed decomposition structure.

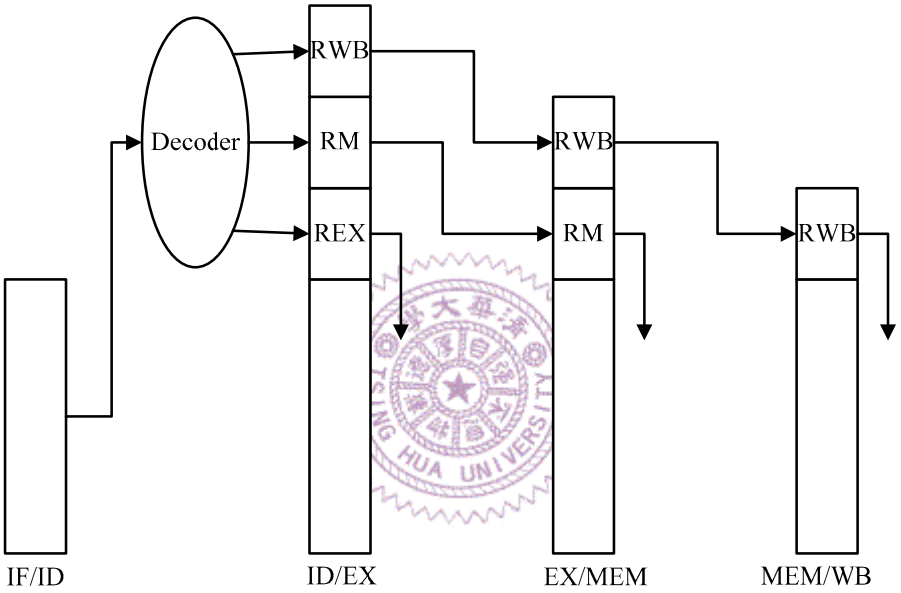

Figure 4.4: Centralized pipelined control-signal generation.

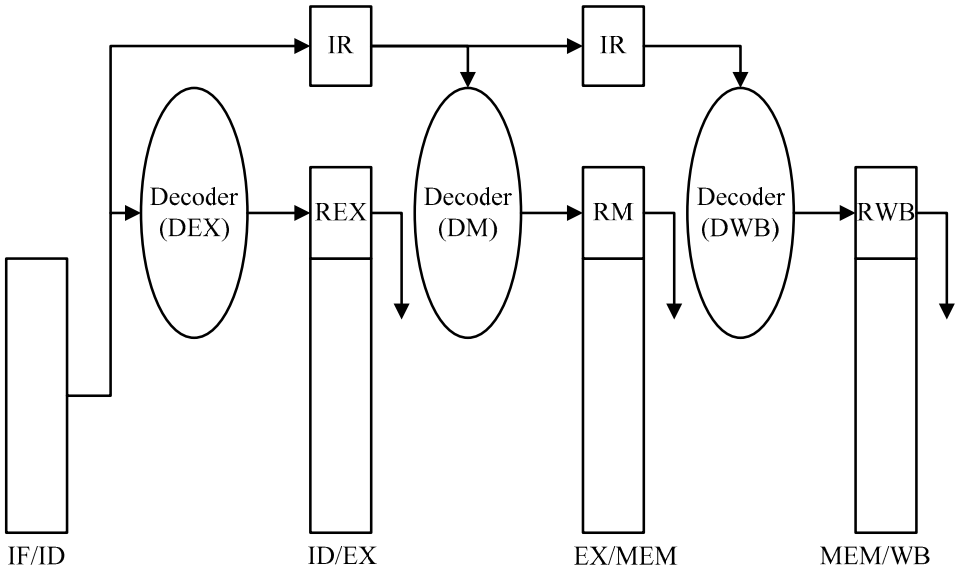

Figure 4.5: Distributed pipelined control-signal generation.

4.2.1 Overview of control-signal generation for a pipelined structure

There are two commonly-used approaches for generating control signals for a pipelined structure. The first [49] decodes all control signals once for all the following stages in the *ID* stage and propagate the control signals down to the pipelined stages, as shown in Figure 4.4 (centralized control-signal decoder). Rather than propagating control signals down to the pipelined stages, the second approach [50] decodes instructions when they are needed at each stage (distributed control-signal decoder), as shown in Figure 4.5. While neither of these approaches is a perfect solution, each has its benefits. For example, the former approach has the advantage of single decoders at the *ID* stage (suitable for designs with less control signals) while the latter approach requires no pipelined registers to store control signals (suitable for designs with large numbers of control signals). In this thesis, we will use the second approach.

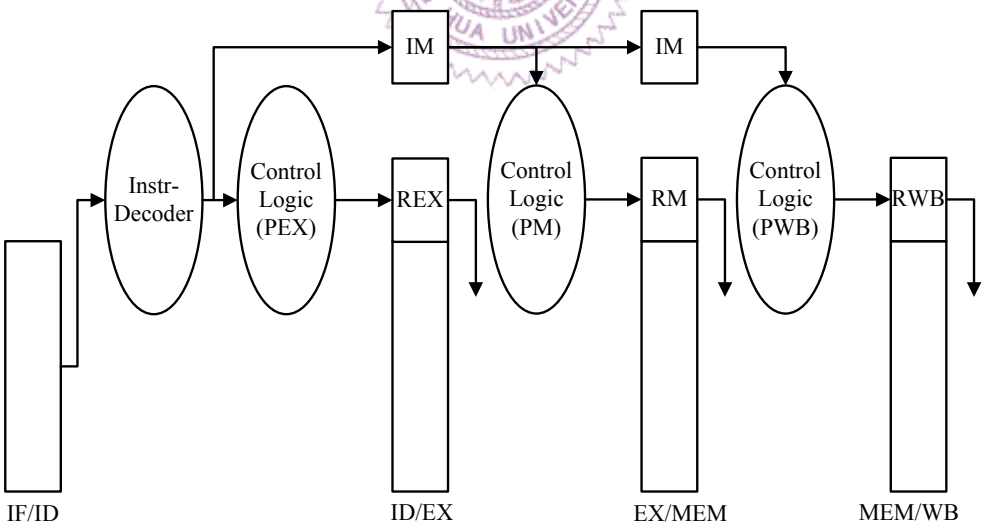

Figure 4.6: The proposed two-stage control-signal decoding structure.

4.2.2 The structure for two-stage pipelined control-signal generation

One disadvantage of a distributed control-signal decoder is that it requires identifying instruction types and generating control signals. To save the logic for identifying instruction types, we propose a two-stage decoding structure where identifying instruction types is performed only once in the decoding stage and generating control signals is performed at all stages. At the decoding stage of the pipeline, in the first stage, an instruction type is identified and an intermediate code is introduced to represent the specific instruction. In the second stage, this intermediate code is decoded to generate the control signals for the *EX* stage. This intermediate code generated at the first stage will be passed along the pipelined stages and used to produce control signals in time for the remaining stages.

Figure 4.6 shows the block diagram of our proposed pipelined instruction decoder. In the first stage, an instruction is decoded by the *Instr-Decoder* and the output is an intermediate code. In the second stage, the generated intermediate code is decoded by the *Control Logic* to generate control signals for the next execution stages. The intermediate code, rather than the long instruction, is also stored in the *Intermediate Code Registers* (*IM*) that will be passed along the pipelined stages.

The generation of intermediate code may incur some control-circuit overhead for instruction identification and control signal generation, depending on how intermediate codes are assigned.

4.2.3 The intermediate-code assignment algorithm

In this section, we present an intermediate code assignment algorithm to reduce the overhead. The goal of the assignment is to simplify the decoder for the generation

of intermediate codes (*Instr-Decoder*) and to simplify the decoders (*PEX*, *PRM*, and *PWB*) for the generation of control signals. To solve the problem, we model the intermediate-code assignment as a state assignment of Finite State Machines (FSMs). We can use the many existing state assignment tools for FSMs to solve our intermediate-code assignment. In this thesis, we use *JEDI* [51] to perform the state assignment.

In an FSM, a state-transition table is used to represent the transitions among states. A row in the table corresponds to a transition. There are four columns in a row: input condition, present state, next state, and output, where input condition and output are binary codes, and present state and next state are symbolic representations. A state-assignment tool will determine the state assignment of an FSM based on the transition table for the overall gain in logic reduction.

Now, we model our intermediate code assignment problem as an FSM transition problem. The transition table construction of an FSM considers two issues: (1) minimizing the logic of *Instr-Decoders* for generating intermediate codes and (2) minimizing the logic of the sub-decoders for generating control signals. To construct the first part of the transition table, we place intermediate codes of instructions with similar instruction op-codes close to each other. First, a *don't care* state, *DC* is created. Then, for each instruction, *IX*, a transition from state *DC* to *IX* under the input condition specified by the op-code of instruction *IX* is created, and all 0s are produced as the transition-outputs. Similarly, to construct the second part of the transition table, we place intermediate codes of instructions generating similar control signals close to each other. To this end, for each instruction *IX*, under any input combination, a transition from *IX* to itself is constructed and control output-signals of the instructions are generated as transition-outputs.

For example, Figure 4.7(a) shows three instructions, *IA*, *IB*, and *IC*, their op-codes, and the generated control signals. Figure 4.7(b) illustrates the state transition table constructed from Figure 4.7(a). The upper part focuses on assigning codes to instructions with similar op-codes (the generation of intermediate code) and the lower part focuses on assigning codes to instructions generating similar control signals. After constructing the state transition table, a state assignment tool is invoked to perform the code assignment.

Instruction	Op-code	Control Signals
IA	011	0101
IB	001	0101
IC	100	1010

(a)

Input	Present State	Next State	Output
011	DC	IA	0000
001	DC	IB	0000
100	DC	IC	0000
xxx	IA	IA	0101
xxx	IB	IB	0101
xxx	IC	IC	1010

(b)

Figure 4.7: (a) Three instructions with their op-codes and output signals, (b) the instruction state transition table (*DC*: don't care state and *x*: input *don't care*).

The quality of an intermediate code assignment also depends on the selected code length. A code length can be a minimum, maximum, or anything in between. Our selection of intermediate codes is determined as follows. From the minimum bit-width that represents all the instructions in the decoder to the maximum bit-width, the bit-width of the original op-code, we generate a two-stage decoder using *JEDI* and estimate its power with the *Synopsys Design Compiler*. The two-stage decoder using the smallest amount of power is selected as our solution.

The proposed horizontal and vertical decompositions can be applied to the design of decoders in various processors. The implementation depends on the Instructions Set Architecture (ISA) and pipelined structure of a processor. ISA designs are categorized into two groups: reduced instruction set computer (RISC) and complex instruction set computer (CISC). For RISC-type ISAs, decoding instructions usually takes one clock cycle (because of simple instruction functions), whereas for the CISC-type ISAs, decoding instructions will take one or more clock cycles because of the complex instruction type. As to the implementation of a pipelined structure, besides the standard pipelined structure, there is a Superscalar (or VLIW) pipeline for high-performance processors where processors fetch and dispatch more than one instruction at a time. Combining these two design options, we have four types of decoders. The first type is a standard RISC pipeline, e.g., *ARM* and *MIPS* processors. The second is a Superscalar RISC pipeline, e.g., *Alpha*, *PA-RISC*, and *MIPS R8000/R10000*, and the third is a standard CISC pipeline, e.g., Intel *486* processors. The final type is a Superscalar CISC pipeline, e.g., *Intel Pentium* processors.

In the standard RISC pipelined structure, the proposed horizontal and vertical decompositions can be directly applied to the structure. In the Superscalar RISC pipeline, the horizontal decomposition can be applied to each control signal decoder in the pipeline. The vertical decomposition can be applied to the instruction decoders in the decoding stage that identifies instruction type and passes the information with the pipelined stages. In the standard CISC pipeline, the instruction decoder decodes a complex instruction into micro-operations (u-ops) and then dispatches each u-op into the following pipelines. As a result, the pipelined stages after the decoding stage are basically the same as the RISC pipelines. To apply the horizontal decomposition, the instruction decoders can be decomposed according to the u-op profiling results. The vertical decomposition is applied to the decoding stage for generating u-ops and

passing the u-op information along the pipeline. Finally, in the Superscalar CISC pipeline, the horizontal decomposition of each decoder in the pipeline can be applied depending on the u-op profiling. The vertical decomposition can also be applied to each decoder in the decoding stage and passing the u-op information along the pipelined stages.

4.3 Experimental results

We have conducted three sets of experiments. In the first experiment, we compare the power consumption and area using the un-decomposed decoder (Figure 4.5) to that using the horizontal decomposed decoder presented in Section 4.1. Next, in the second experiment, we compare the power consumption and area using the un-decomposed decoder (Figure 4.5) to that using the vertical decomposed decoder presented in Section 4.2. Finally, in the third experiment, we examine the power consumption and area overheads by applying both horizontal and vertical decomposition techniques to the decoder. In this experiment, we used the *Powerstone* as our benchmarking programs.

The benchmarking circuits are described in *Verilog*. The final designs were generated by the *Synopsys Design Compiler* using the *TSMC* 0.25um cell library. For all experiments, we used *Synopsys PrimePower* to compute the power consumption (both dynamic and static power consumption). The output capacitances of the instruction decoder and processor are set to the input capacitance of D flip-flop (0.06 pF) and output PAD (0.10 pF), respectively. The goal of optimization is area. The timing and area data were reported by *Design Compiler* with the *report_timing* and *report_area* options.

We used *ARMSDT v2.5* to compile each benchmark program into a machine code. The generated machine code and our *ARM Verilog* code were fed into *PrimePower* for

calculating power consumption. Table 4.1 shows the power consumption of the un-decomposed instruction decoder and processor. In this table, *DEX*, *DM*, and *DWB* are the results for the instruction decoders in the *ID, EX,* and *MEM* stages, respectively. *DEX*, *DM*, and *DWB* decode control signals for the first execution stage, execution stages (for multiple execution-stage instructions), memory access, and write-back, respectively. *IR* is the instruction register for storing instruction information (op-code), and *PE* is the processor. The results show that the instruction decoder (*DEX*, *DM*, *DWB*, and *IR*) consumes about 9.83% of the total processor power consumption. Table 4.2 shows the timing and area of the un-decomposed decoder.

Table 4.1: Power consumption of the un-decomposed decoder and processor.

Benchmark	DEX(W)	DM(W)	DWB(W)	IR(W)	PE(W)
adpcm	1.29E-04	2.29E-04	1.53E-04	7.83E-05	5.07E-03
auto2	9.17E-05	1.63E-04	1.09E-04	8.68E-05	5.77E-03
bcnt	1.16E-04	2.08E-04	1.38E-04	9.43E-05	5.06E-03
bilinear2	1.17E-04	2.16E-04	1.41E-04	9.46E-05	5.33E-03
bilv	1.17E-04	2.08E-04	1.39E-04	8.57E-05	4.83E-03
binary	1.04E-04	1.75E-04	1.23E-04	7.96E-05	4.98E-03
blit	1.08E-04	2.27E-04	1.35E-04	9.42E-05	5.97E-03
brev	6.85E-05	1.22E-04	8.12E-05	7.40E-05	5.16E-03
compress	9.70E-05	1.74E-04	1.07E-04	7.06E-05	6.25E-03
crc	1.03E-04	1.83E-04	1.22E-04	8.91E-05	5.15E-03
des	9.10E-05	1.79E-04	1.20E-04	8.80E-05	5.15E-03
dhry	1.04E-04	1.85E-04	1.24E-04	7.85E-05	5.52E-03
engine	9.73E-05	1.73E-04	1.16E-04	8.18E-05	5.06E-03
fir_int	9.80E-05	1.53E-04	1.08E-04	4.92E-05	5.10E-03
g3fax	1.06E-04	2.12E-04	1.42E-04	9.07E-05	5.57E-03
g721	1.36E-04	2.34E-04	1.62E-04	8.36E-05	5.39E-03
insert	1.06E-04	1.90E-04	1.26E-04	8.67E-05	5.66E-03
jpeg	1.12E-04	2.17E-04	1.45E-04	8.75E-05	5.47E-03
ludcmp	9.83E-05	1.75E-04	1.17E-04	8.43E-05	4.89E-03
mattst	1.50E-04	2.68E-04	1.58E-04	7.62E-05	5.10E-03
pocsag	1.39E-04	2.28E-04	1.65E-04	9.66E-05	5.76E-03
select	2.81E-06	5.18E-06	3.45E-06	4.79E-05	2.35E-03
summin	1.23E-04	2.19E-04	1.46E-04	1.11E-04	6.07E-03
ucbqsort	1.10E-04	1.87E-04	1.31E-04	6.20E-05	4.39E-03
v42	1.18E-04	2.11E-04	1.41E-04	8.97E-05	1.95E-03
whetstone	9.80E-05	1.29E-04	1.03E-04	8.09E-05	5.10E-03
Average	1.05E-04	1.87E-04	1.25E-04	8.24E-05	5.08E-03

Table 4.2: Timing and area information of the un-decomposed decoder and processor.

	DEX	DM	DWB	IR	PE
Area	5418	16332	9936	2841	307054
Timing(ns)	24.62	32.87	30.93	0.76	50

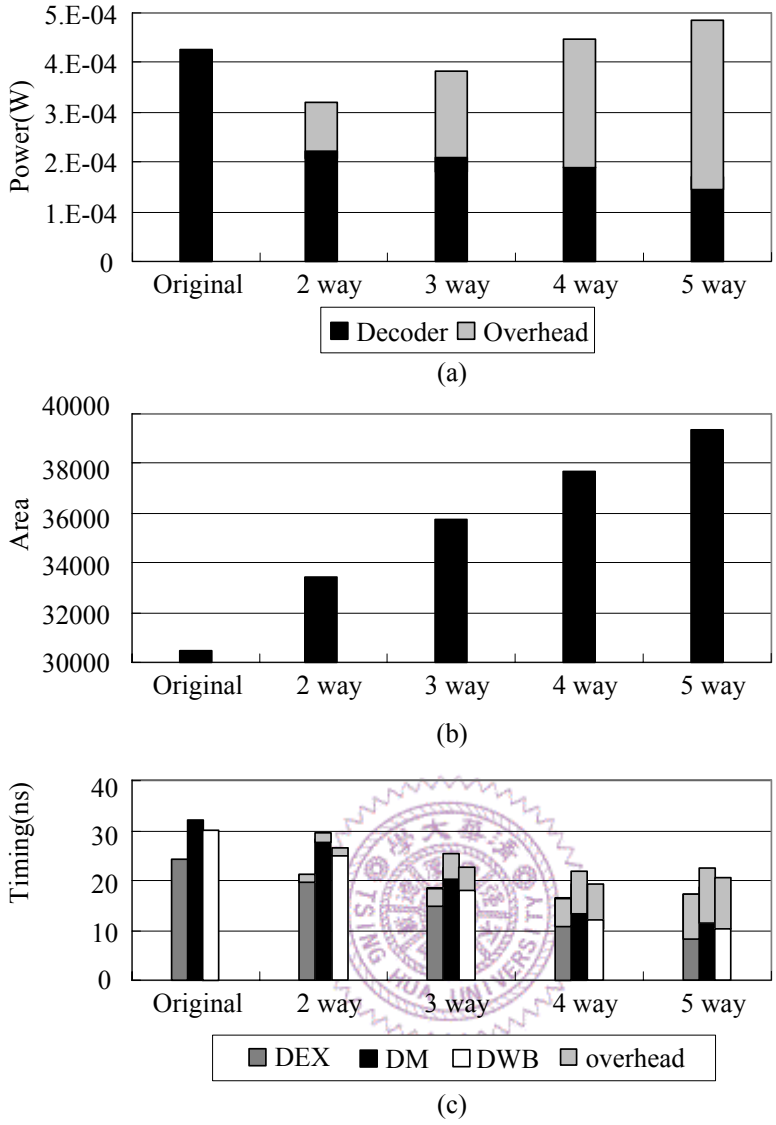

Figure 4.8: The comparisons of (a) power consumption, (b) area, and (c) timing for multi-way partitions.

4.3.1 The horizontal decomposition

In the first experiment, we evaluated the power reduction, area overhead, and timing reduction of our proposed decomposed instruction decoding method described in Section 4.1. The instruction transition graph is constructed from the profiling results of the *Powerstone* benchmarks.

Figure 4.8 shows the average power consumption, area, and timing comparisons both on instruction decoders without decomposition and decoders with different

number of partitions. Figure 4.8(a) shows the power consumption, which is the sum of *DEX*, *DM*, and *DWB*. The results show that when the decoder is partitioned into more than two partitions, the overhead of the turning-off control circuit consumes more power. As a result, the two-way partitioning achieves the best power reduction (23.14% power reduction).

Figure 4.8(b) shows the results of area, which is the sum of *DEX*, *DM*, and *DWB*. The results show that the area of the partitioned circuit is proportional to the number of partitions. Figure 4.8(c) shows the timing comparisons. The results show that the timing of the decomposed decoder is reduced because the original decoder is decomposed into several smaller and parallel decoders. The results also show that timing is no longer reduced when the number of partitions becomes larger than four.

In summary, Figure 4.8 shows that when the partition number is two, our proposed instruction-decoding method achieves on an average of 23.14% in power reduction as compared to the un-decomposed instruction decoder. The results also show that the critical timing of decomposed decoder (*DM*) is 6.53% shorter than the original decoder and the area overhead of the decomposed decoding circuits is 8.93% larger than the original decoding circuit. The results demonstrate that the horizontal decoding technique can improve the power consumption and critical timing of instruction decoders with small area overhead.

As to the running time of the partitioning program, we have conducted an experiment to record the number of iterations tried. Table 4.3 shows the results. From this table, we can see that the number of iterations is reasonably small.

Table 4.3: The iteration number of the partitioning algorithm.

	2 way	3 way	4 way	5 way
Iteration number	1378	2652	3675	4704

4.3.2 The vertical decomposition

In this experiment, we re-designed the instruction decoder (Figure 4.5) into a two-stage decoder (Figure 4.6) where the intermediate codes were assigned using the algorithm described in Section 4.2.3. The option we used for generating intermediate code was *jedi –e r*. Using the option and the FSM transition model, we generated intermediate codes with different bit-widths ranging from eight (the smallest bit-width to represent all 142 instructions) to sixteen (the original op-code bit-width). For each bit-width, we generated a two-stage decoder using the timing of the un-decomposed decoder shown in Table 4.2 as the timing constraint. Since the smallest estimated power is the bit-width of ten, we used ten bits as our intermediate code length to generate a two-stage instruction decoder.

Figure 4.9 shows the average power consumption, area, and timing comparisons of one- and two-stage instruction decoding. In the figure, *Instr-Decoder*, *PEX*, *PM*, *PWB*, and *IM* are the sub-decoders shown in Figure 4.6. Figures 4.9(a) and 4.9(b) show that the power consumption and area of the *ID* stage are increased. This happens because the intermediate code is synthesized under the timing constraint of the un-decomposed decoder (24.62 ns, Table 4.2). On the other hand, the power consumption, area, and timing of *PM* and *PWB* are reduced, due to the removal of the instruction identifying circuit. The pipelined registers for propagating instruction information are also reduced, because the intermediated code (ten bits) is shorter than the original op-code (sixteen bits). As a result, the two-stage decoding structure achieves 14.73% in power reduction and 5.14% in area reduction.

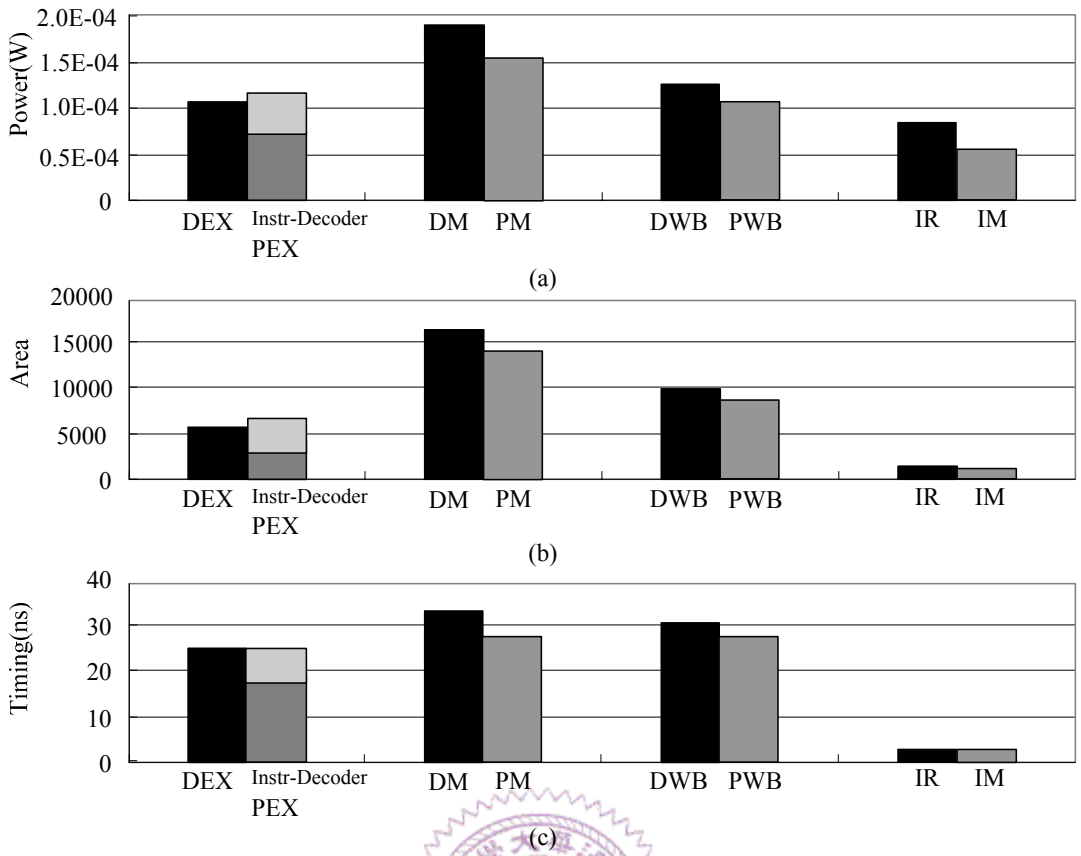

Figure 4.9: The comparisons of (a) power consumption, (b) area, and (c) timing for one- and two-stage instruction decoding.

4.3.3 Combining horizontal and vertical decompositions

In this experiment, we apply both techniques (horizontal and vertical decomposition) to the pipelined instruction decoder. First, we applied the two-way horizontal decomposition. Then, for each sub-decoder, we applied the vertical decomposition. For the two sub-decoders, the bit-widths of the intermediate codes were seven and ten, respectively. As a result, he bit width of *IM* is ten.

Figure 4.10 compares the average power consumption, area, and timing of the original decoder with decoders containing horizontal and vertical decompositions. Figure 4.10(a) shows that power consumption reduces by up to 34.28% by applying both techniques to the original instruction decoder. The results also show that the area

overhead of the decomposed decoding circuits is 7.85% larger than that of the original decoding circuit, while the critical timing of the decomposed decoder (*DM*) is 11.36% shorter than the original one.

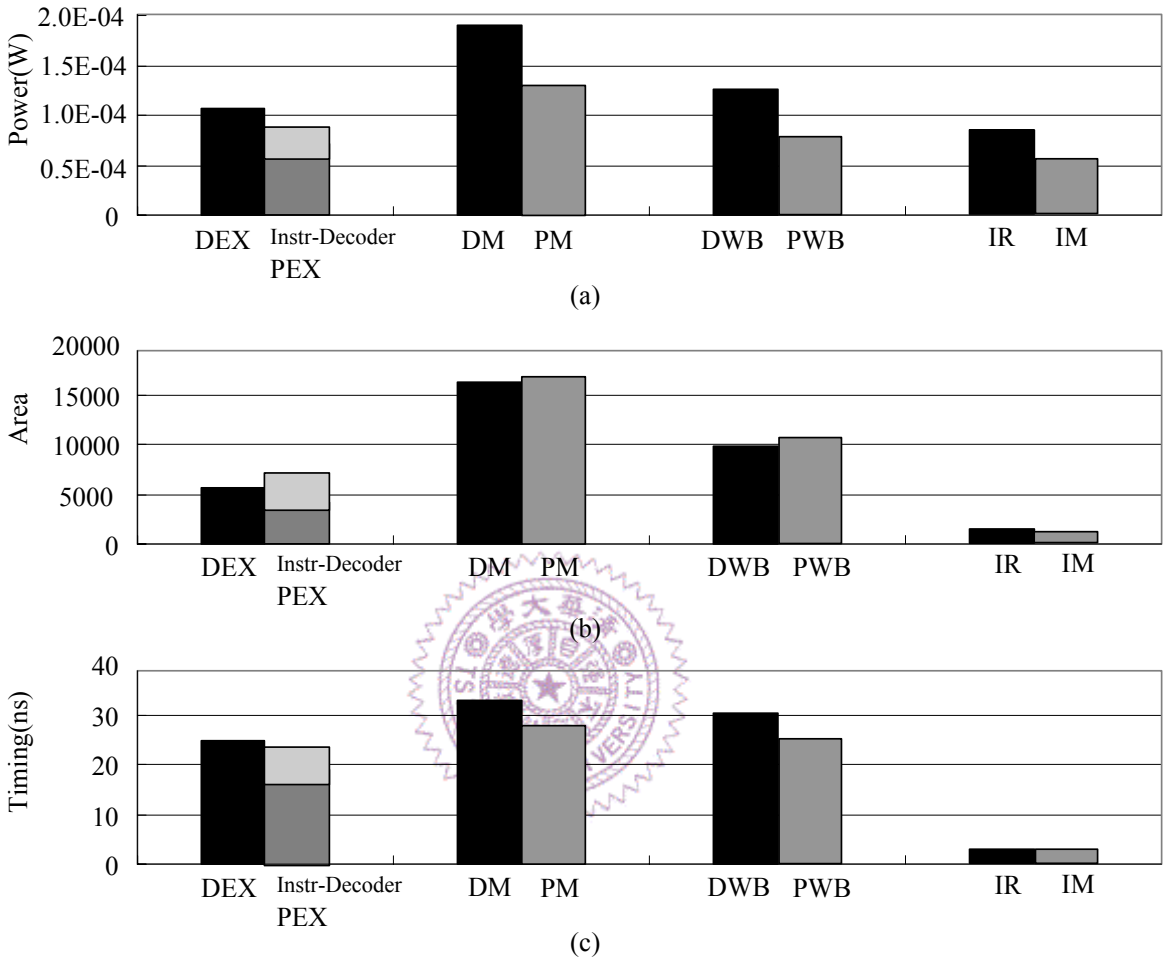

Figure 4.10: The comparisons of (a) power consumption, (b) area, and (c) timing by applying both horizontal and vertical decompositions.