Journal of the Korea Academia-Industrial cooperation Society
Vol. 16, No. 1 pp. 639-645, 2015

http://dx.doi.org/10.5762/KAIS.2015.16.1.639
ISSN 1975-4701 / eISSN 2288-4688

Niod II 코어기반 가속기 비교

송기용[1*]
[1]충북대학교 전자공학부

Comparison of Nios II Core-based Accelerators

Gi-Yong Song[1*]
[1]**School of Electronics Enginneering, Chungbuk National University**

요 약 Nios II 코어에 기반한 하드웨어 가속기를 checksum과 residue checking 알고리즘을 대상으로 하여 HDL 코딩으로 해당 하드웨어를 구현하는 component 방식, 프로세서 명령어세트 확장에 의한 custom instruction 방식과 C2H 컴파일러로 해당 로직을 자동 생성하는 C2H 방식으로 구현하고, 실행 결과를 분석 및 비교한다. 비교 결과 실행 소요시간 기준의 경우 C2H 방식 구현이 최단시간 수행을, 그리고 하드웨어 추가 소요량 기준의 경우 custom instruction 방식 구현이 최소의 하드웨어를 추가로 사용함을 확인한다.

Abstract Checksum and residue checking accelerators were implemented on a Nios II core-based platform according to component method, in which the corresponding hardware was implemented with HDL coding, a custom instruction method, in which the instruction set of the processor was extended, and the C2H method, in which the corresponding logic was automatically created by the C2H compiler. The processing results from each accelerator for each algorithm were then examined and compared. The results of the comparison showed that the accelerator implemented with the C2H method is the fastest in terms of the execution time, and the accelerator with custom instruction requires the least add-on from the viewpoint of add-on hardware.

Key Words : accelerator, checksum, Nios II core, residue checking

1. 서론

SOPC(System-On-a-Programmable-Chip) 기술은 다양한 IP들을 선택, 구성하는 top-down 설계와 검증방식으로 하나의 시스템을 구현하고 이를 통하여 first-time-silicon을 추구한다. SOPC 개발 과정은 일반적으로 하드웨어부분과 소프트웨어부분으로 나뉜다. 하드웨어부분 설계는 Nios II 시스템을 생성하는 것, 즉 어플리케이션을 실행할 수 있는 회로를 구성하는 것이고, 소프트웨어부분 설계는 시스템 내장 API와 어플리케이션을 C/C++언어를 이용하여 설계하는 것이다[1]. Nios II 코어기반 SOPC 개발에는 Altera사의 Quartus II, SOPC builder, Nios II IDE 등의 소프트웨어가 사용된다. Quartus II로 프로젝트를 생성하고, SOPC builder로 Nios II 시스템을 구성한 후, component 방식, custom instruction 방식, 또는 C2H 방식으로 IP를 설계하고, 검증한 후 시스템에 탑재한다. 끝으로 응용프로그램을 생성하고, Nios II IDE를 이용하여 어플리케이션을 구현된 시스템에서 수행시킨다[2]. 본 논문에서는 checksum 가속기와 residue checking 가속기를 각각의 방식으로 구현하고, 시스템에 탑재 및 수행하여 동작을 확인하고 각 구현 경우의 실행시간 및 소요 logic을 비교하여 사양에 따른 구현방식 선택시의 참고를 제시한다.

이 논문은 2013년도 충북대학교 학술연구지원사업의 연구비 지원에 의하여 연구되었음
*Corresponding Author : Gi-Yong Song(Chungbuk National University)
Tel : +82-43-261-2452 email : gysong@cbnu.ac.kr
Received July 24, 2014 Revised (1st September 29, 2014, 2nd October 30, 2014, 3rd November, 10, 2014) Accepted January, 8, 2015

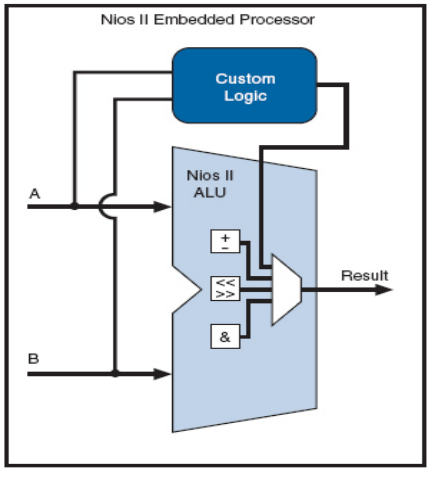
2. Nios II 코어기반 가속기 구현

Altera Nios II SOPC 플랫폼은 다음과 같은 세가지 방식으로 가속기 등의 IP 설계를 지원한다.

1) 해당로직 component를 HDL로 코딩하여 시스템에 탑재하는 방식,
2) Nios II 프로세서에 custom instruction을 도입하는 방식,
3) Nios II C2H(C-to-Hardware Acceleration) 컴파일러를 사용하여 해당회로를 생성하고 시스템에 추가하는 방식.

Component 방식은 VHDL 또는 Verilog HDL을 이용하여 해당로직을 설계 및 검증한 후 component로 등록하고, SOPC builder를 이용하여 Nios II SOPC 시스템에 탑재하여 가속기를 구현한다. Component는 시스템 interconnect fabric인 Avalon Switch Fabric으로 Nios II 프로세서 또는 다른 component에 연결된다[3].

Custom instruction 방식의 경우 VHDL 또는 Verilog HDL을 이용하여 해당 명령어수행 하드웨어인 custom 로직을 코딩하고 검증한 후, SOPC builder를 이용하여 custom instruction을 Nios II 프로세서 명령어세트에 추가한다. 그림 1은 Nios II 프로세서의 ALU에 추가된 custom 로직 블록을 보여준다[4].



[Fig. 1] Block diagram of Nios II custom instruction

Nios II C2H 컴파일러는 ANSI C 코드로부터 가속기 하드웨어의 HDL 코드를 생성하고, 생성된 하드웨어를 Nios II SOPC 시스템에 탑재하는 툴로, C2H로 생성된 하드웨어는 다른 주변장치와 마찬가지로 Avalon Switch Fabric 으로 시스템에 연결된다. C2H 방식의 경우, 가속기 하드웨어가 C2H 컴파일러에 의해 자동으로 생성, 탑재됨으로 빠르고 간편하게 가속기를 구현할 수 있으나, 최적화된 구현을 위해서는 알고리즘의 논리구성, 메모리 구성, 데이터의존성 축소 및 리소스 공유 등에 대한 분석을 통한 구현대상 함수 정의 C 코드의 재구성이 필요하며 아직 사용에서 많은 제한을 받고 있다[5].

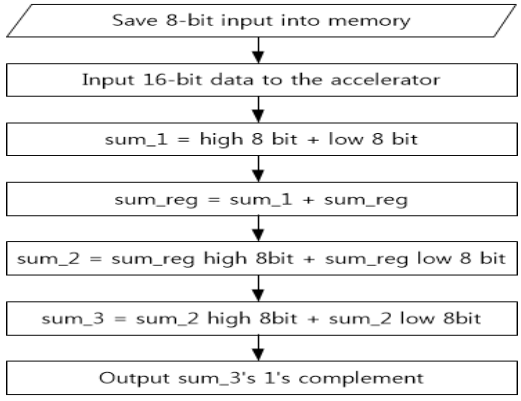
(1) C++문법을 지원하지 않는다.
(2) Struct and Union 구조를 지원하지 않는다.
(3) Recursive 구문을 지원하지 않는다.
(4) 메모리 동적할당을 지원하지 않는다.

본 논문에서는 데이터의 정확성 검증에 자주 사용되는 checksum과 residue checking을 각 구현방식에 따라 가속기로 구현하고 실행결과를 비교하였다.

3. 가속기 구현

3.1 Checksum 가속기 구현

자료전송에서의 오류발생여부를 확인하는 checksum을 그림 2 와 같이 설정하고 가속기를 구현하였다.



[Fig. 2] Checksum flow chart

설정한 checksum의 C 코드 구현을 아래에 보였으며, 이 C 코드의 수행소요시간은 가속기 도입에 따른 성능비교의 기준이 된다.

```
void sw_checksum(int *input_data, int
*output_data,...)
{ ...
  for (i = 0; i < N; i++)
  {
    sum_1 = (input_data[i] & (0x000000ff)) +
((input_data[i] >> 8) & (0x000000ff));
    sum_reg += sum_1;
    sum_2 = (sum_reg & (0x000000ff)) + ((sum_reg
>> 8) & (0x000000ff));
    sum_3 = (sum_2 & (0x000000ff)) + ((sum_2
 >> 8) & (0x000000ff));
    output_data[i] = (~(sum_3 & (0x000000ff))) &
(0x000000ff);
  }
}
```

3.1.1 Component 방식 구현

가속기 로직의 HDL작성[6], component 등록 및 Nios II 코어기반 시스템에 탑재의 순서로 진행되며, 작동시 main()은 아래에 보인 component_checksum() 을 호출하고, component_checksum() 은 IORD_32DIRECT() 와 IOWR_32DIRECT() 호출하여 가속기 component에 접근한다.

```
void component_checksum(int *input_data, int *
output_data, int length)
{
...
for (i=0; i< N; i++)
{
IOWR_32DIRECT(COMPONENT_BASE,0,input_data[i]);
output_data[i] = IORD_32DIRECT(COMPONENT_BASE,
0);
}
}
```

3.1.2 Custom instruction 방식 구현

Custom instruction 방식은 component 코드에 wrapping을 적용하여 구현한다. 본 논문에서의 custom instruction은 Extended 방식으로 설정하였고, opcode를 가리키는 N은 2비트로 설정하여 N = 1일 때 32 비트 쓰기동작을, N = 2 일 때 읽기동작을 수행한다. Wrapping 코드를 작성할 때, 입력데이터가 구동되면 몇 개의 클럭 사이클 뒤에 출력데이터가 발생되므로 대응되는 제어신호들을 출력데이터의 발생시간만큼 지연시켜야 한다. 이와같이 Verilog HDL 코드를 작성하고 SOPC builder에 있는 Nios II 프로세서 명령어 세트에 custom instruction을 추가한다. 생성된 system.h에서 아래와 같이 사용자 명령어함수가 선언되어 있는 것을 확인할 수 있다.

```
#define ALT_CI_CHECKSUM_CI_INST_N 0x00000000
#define ALT_CI_CHECKSUM_CI_INST_N_MASK ((1<<2)-1)
#define ALT_CI_CHECKSUM_CI_INST(n,A) __builtin_
custom_ini(ALT_CI_CHECKSUM_CI_INST_N+(n&ALT_CI_C
HECKSUM_CI_INST_N_MASK),(A))
```

main() 코드는 사용자 설정명령어를 실행하는 ci_checksum() 함수를 호출하여 custom instruction 방식으로 구현된 가속기를 작동시킨다.

```
void ci_checksum( int * input_data, int *
output_data, int length)
{
  ...
  for (k = 0; k < N; k++)
   {

    ALT_CI_CHECKSUM_CI_INST(1,(int*)input_dat
a[k]);
    output_data[k] = ALT_CI_CHECKSUM_CI_INST
(2, 0);
   }
}
```

3.1.3 C2H 방식 구현

가속기 구현대상은 아래의 함수 원형으로 선언되는 c2h_checksum()으로 sw_checksum()과 동일한 함수 정의를 가진다.

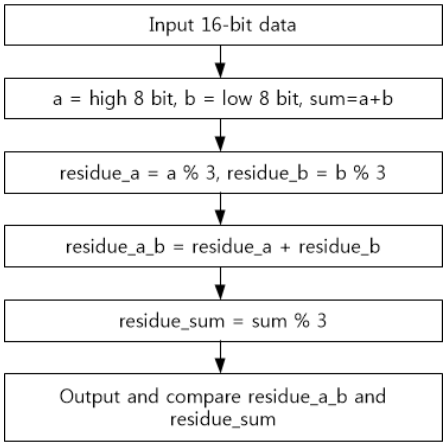
```
void c2h_checksum(volatile int * __restrict__
input_data, int * __restrict__ output_data, int
length)
{
 ...
}
```

Nios II IDE C2H 컴파일러는 어플리케이션을 다시 build하여 하드웨어 가속기를 HDL코드로 생성하고, SOPC 시스템에 탑재한다. 다음은 자동으로 컴파일 되어 생성된 파일명이다.

- accelerator_checksum_c2h_checksum.v
- accelerator_checksum_c2h_checksum_managed_instance.v
- c2h_port_byte_refine.v

3.2 Residue checking 가속기 구현

가산연산에서의 오류검출을 위한 residue를 그림 3과 같이 설정하고 해당 가속기를 구현하였다[7].



[Fig. 3] Residue checking flow chart

설정한 residue checking의 C 코드 구현을 아래에 보였으며 이 C코드의 수행소요시간은 가속기 도입에 따른 성능비교의 기준이 된다.

```
void sw_residue(int *input_data, int *output_data, ...)
{
  ...

  for (i = 0; i < N; i++)
  {
     a = (input_data[i] >> 8) & (0x000000ff);
     b = input_data[i] & (0x000000ff);
     sum = a + b;
     residue_a = a %3;
     residue_b = b %3;
     residue_a_b = (residue_a + residue_b) % 3;
     residue_sum = sum % 3;
     output_data[i] = ((residue_a_b & (0x0000000f)) << 2) + (residue_sum & (0x00000000f));
  }
}
```

3.2.1 Component 방식 구현

구현과정은 checksum 가속기의 경우와 동일하며 main()은 아래의 component_residue()를 호출하여 탑재된 가속기에 접근한다.

```
void component_residue(int *input_data, int *output_data, int length)
{
...
for (i=0; i< N; i++)
{
IOWR_32DIRECT(COMPONENT_BASE,0,input_data[i]);
output_data[i] = IORD_32DIRECT(COMPONENT_BASE,0);
}
}
```

3.2.2 Custom instruction 방식 구현

checksum 가속기의 custom instruction 방식 구현의 경우와 동일한 과정을 따르며, system.h에서 아래와 같이 사용자 명령어 함수가 선언되어 있는 것을 확인할 수 있고, main()은 ci_residue() 함수를 호출하여 custom instruction을 수행한다.

```
#define ALT_CI_RESIDUE_CI_INST_N 0x00000000
#define ALT_CI_RESIDUE_CI_INST_N_MASK ((1<<2)-1)
#define ALT_CI_RESIDUE_CI_INST(n,A) __builtin_custom_ini(ALT_CI_RESIDUE_CI_INST_N+(n&ALT_CI_RESIDUE_CI_INST_N_MASK),(A))
```

```
void ci_residue( int * input_data , int * output_data, int length)
{
  ...
  for (k = 0; k < N; k++)
   {
    ALT_CI_RESIDUE_CI_INST(1,(int*)input_data[k]);
    output_data[k] = ALT_CI_RESIDUE_CI_INST(2, 0);
   }
}
```

3.2.3 C2H 방식 구현

C2H 컴파일러가 생성하게 될 가속기의 구현함수는 sw_residue()와 동일한 함수 정의를 가지고 아래의 원형으로 선언되는 c2h_residue()이며,

```
void c2h_residue(volatile int * __restrict__
input_data, int * __restrict__ output_data, int
length)
{
 ...
}
```

다음은 자동으로 컴파일 되어 생성된 파일을 보여준다.

- accelerator_residue_c2h_residue.v
- accelerator_residue_c2h_residue_managed_instance.v
- c2h_port_byte_refine.v

4. 가속기 실행 및 비교

그림 4는 checksum 처리의 software구현 및 component 방식, custom instruction 방식과 C2H 방식 가속기 구현시 실행 소요시간이 1397 us -> 330 us -> 284 us -> 119 us 로 단축되어, software구현을 기준으로 하는 speed-up factor가 4.24 -> 4.92 -> 11.78 로 나타남을 보여준다.

```
nios2-terminal: connected to hardware target
using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device 1,
instance 0
nios2-terminal: (Use the IDE stop button or
Ctrl-C to terminate)

All the hardware and software results match
Processing times:
Software processing time was: 0.001397 seconds
Component processing time was: 0.000330 seconds
Custom Instruction processing time was: 0.000284
seconds
C2H processing time was: 0.000119 seconds

Component versus software speed-up factor was:
4.24 times
Custom Instruction versus software speed-up
factor was: 4.92 times
C2H versus software speed-up factor was: 11.78
times
Exiting...
nios2-terminal: exiting due to ^D on remote
```

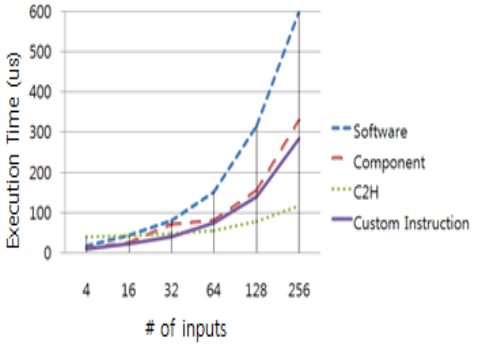
[Fig. 4] Processing results from checksum accelerator

그림 5는 residue checking 처리의 software구현 및 component 방식, custom instruction 방식과 C2H 방식 가속기 구현시 실행 소요시간이 600 us -> 330 us -> 284 us -> 117 us 로 단축되어, software구현을 기준으로 하는 speed-up factor가 1.82 -> 2.11 -> 5.12 로 나타남을 보여준다. checksum 과 residue checking 가속기 실행시간과 software 처리 실행시간의 비교에서 가속기 구현시의 성능 개선효과를 확인할 수 있다.

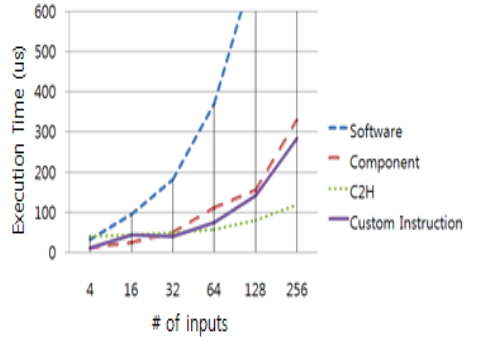
```
nios2-terminal: connected to hardware target
using JTAG UART on cable
nios2-terminal: "USB-Blaster [USB-0]", device
1, instance 0
nios2-terminal: (Use the IDE stop button or
Ctrl-C to terminate)

All the hardware and software results match
Processing times:
software processing time was: 0.000600 seconds
Component processing time was: 0.000330 seconds
Custom Instruction processing time was:
0.000284 seconds
C2H processing time was: 0.000117 seconds

Component versus software speed-up factor was:
1.82 times
Custom Instruction versus software speed-up
factor was: 2.11 times
C2H versus software speed-up factor was: 5.12
times
Exiting...
nios2-terminal: exiting due to ^D on remote
```

[Fig. 5] Processing results from residue checking accelerator

그림 6, 그림 7은 checksum과 residue checking 각각의 경우 입력의 갯수가 커짐에 따른 software와 3가지 구현방식의 실행결과를 그래프로 보여준다. checksum과 residue checking 가속기 모두 component 방식 -> custom instruction 방식 -> C2H 방식 구현의 순서로 실행시간이 짧아진다.

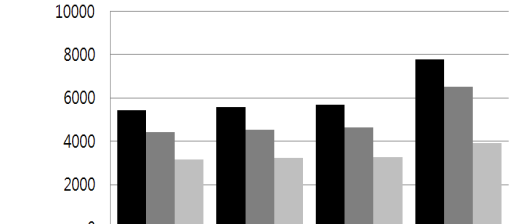


[Fig. 6] Processing time for checksum accelerator



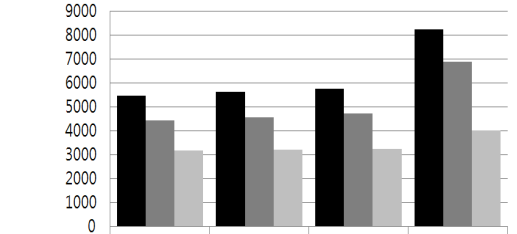
[Fig. 7] Processing time for residue checking accelerator

그림 8, 그림 9는 checksum과 residue checking 가속기를 각 방식으로 구현한 경우의 소요 logic elements를 보여준다.



	Software	Custom Instruction	Component	C2H
Logic Elements	5433	5556	5653	7758
Combinational	4418	4500	4604	6490
Registers	3153	3224	3251	3911

[Fig. 8] Logic element utilization for checksum accelerator



	Software	Custom Instruction	Component	C2H
Logic Elements	5441	5617	5758	8219
Combinational	4426	4561	4710	6862
Registers	3153	3203	3230	3994

[Fig. 9] Logic element utilization for residue checking accelerator

그림 8, 그림 9에서 보이는 것과 같이 checksum과 residue checking 가속기 경우 모두 custom instruction 방식 -> component 방식 -> C2H 방식 구현의 순서로 추가 하드웨어 logic elements 소요량이 증가하며, 이는 타깃 칩의 제공 logic에 가속기가 수용 가능한지 등 칩 선택의 기준으로 사용된다.

5. 결 론

본 논문에서는 checksum과 residue checking 처리 하드웨어 가속기를 Nios II 코어기반 플랫폼을 이용하여 component 방식, custom instruction 방식 및 C2H 방식으로 구현하고 실행결과를 비교하였다.

두 경우의 가속기 모두에서

(1) component 방식 -> custom instruction 방식 -> C2H 방식의 순서로 실행시간이 짧아지므로 실행 소요시간 단축이 우선시되는 구현사양의 경우 C2H 방식으로 가속기를 구현하는 것이 적합하고,

(2) custom instruction 방식 -> component 방식 -> C2H 방식의 순서로 추가 소요 logic elements가 증가하므로 하드웨어 추가 소요량의 최소화가 우선시되는 구현사양의 경우 custom instruction 방식을 사용하는 것이 적합하다.

References

[1] Li Junwei, Yan Han, "The Development of a SOPC system

based Nios II," Sciencepaper Online, 2007.
[2] Yu-Chih Liu, M, Hardware objects & accelerators on Nios II platform, Thesis for Master of Science, Department of Computer Science and Engineering ,Tatung University, 2009.
[3] Altera Corporation, "Avalon Interface Specifications," http://www.altera.com/literature/manual/mnl_avalon_spec.pdf,May. 2013.
[4] Altera Corporation, "Nios II Custom Instruction User Guide," http://www.altera.com/literature/ug/ug_nios2_custom_instruction.pdf, January. 2011.
[5] Altera Corporation, "Nios II C2H Compiler User Guide," http://www.altera.com/literature/ug/ug_nios2_c2h_compiler.pdf, November. 2009.
[6] Bob Zeidman Verilog Designer's Library, Prentice Hall PTR, pp. 261 - 268, 1999.
[7] Douglas J. Smith HDL Chip Design: A Practical Guide for Designing, Synthesizing & Simulating Asics & FPGAs using VHDL or Verilog, Doone Publications, pp. 279 - 312, 1999.

송 기 용(Gi-Yong Song) **[정회원]**

- 1978년 2월 : 서울대 공대 공교과 (학사)
- 1980년 2월 : 서울대 대학원 전자과 (석사)
- 1983년 3월 ~ 현재 : 충북대 교수
- 1995년 12월 : 미) University of Louisiana, Lafayette (박사) 컴퓨터공학

<관심분야>
디지털논리 설계 및 검증, SoC 설계