디지털논리회로설계 2 – Term Project

"꿈을 가지십시오. 그리고 정열적이고 명예롭게 이루십시오."

Factorial computation system

2023. 11. 30.

Changelogs

Version	Date	Description
1.0	2023.11.07.	Initial release
2.0	2023.11.20.	- Byte addressing 에 따른 address region 및 offset 수정 (12p, 13p, 16p, 18p) - Factorial core 의 operator 제한 범위 명확화 - 동작 설명의 operand 범위 오타 수정 - Quartus project 파일은 선택적으로 제출하도록 명시 - 감점사유 추가 (컴파일 관련)
3.0	2023.11.30.	- opclear 가 opclear, intrEn, operand 를 제외한 나머지만 초기화하도록 수정 - opclear 는 clear 직후 testbench 가 직접 0 으로 만들어야 함을 명시 - 연산 시작 전 혹은 연산 도중 opclear 발생은 가정하지 않음 추가 - Factorial 연산의 종료 조건 모호함을 아래와 같이 수정 (4p, 6p, 12p) 기존: 매 곱셈이 수행된 후에는 operand 가 1 이 될 때까지 1 씩 감소하여 factorial 연산을 수행할 수 있도록 한다. 변경: 매 곱셈이 수행된 후에는 operand 가 1 이 될 때까지 1 씩 감소하며, operand 가 1 이 되면 더 이상 곱셈을 수행하지 않고 연산을 종료한다. (즉, operand 가 2 일때 까지만 곱셈을 한다) (자세한 수정 사항은 해당 페이지를 읽어볼 것) - operand register 관련 구현 방법 추가 (8p) - BUS 에서 *m_req* 가 0 인 경우 주소가 맞더라도 select signal 이 발생하지 않도록 수정 (13p) - 기타 오타 수정

Table of Contents

1. Overview

본 문서는 2023 학년도 2 학기 디지털논리회로 2 과목에서 학생들이 설계 및 구현해야 할 프로젝트의 주제, 사양 등을 기술한다. 학생들은 이 문서를 자세히 읽고 이해한 후 프로젝트를 수행하도록 한다.

이번 프로젝트의 주제는 “Factorial computation system”이다. 이 시스템은 factorial 연산을 사용자가 원하는 대로 수행한다.

구체적인 설계 사양은 업데이트될 수 있습니다. 업데이트될 때마다 공지사항에 공지할 테니 확인하기 바랍니다.

참고사항: Port/module 이름은 “Courier New” font 를 사용하며 register 이름은 *Italic font* 를 사용한다.

2. Feature summary

2.1. Factorial

- Operand n = 0, 1, 2, 3, … 2^{31} - 1 일 때 $n!$ 의 계산을 수행한다 (즉, n 의 범위는 32bits signed integer 의 양수 표현 범위와 같다)
- Factorial core 는 내부의 multiplier 를 통해 64bits X 64bits 의 곱셈을 수행하여 128bits 결과를 생성한다. 128bits 의 곱셈 결과는 high 64 bits, low 64 bits 로 나누어 *result_h, result_l* 에 각각 저장된다.
- Factorial 연산은 *operand* 와 *result_l* 을 곱하여 수행하도록 한다. 단, *result_l* == 64’d0 인 경우 *operand* 와 *result_h* 를 곱하도록 한다.
- 매 곱셈이 수행된 후에는 *operand* 가 1 이 될 때까지 1 씩 감소하며, *operand* 가 1 이 되면 더 이상 곱셈을 수행하지 않고 연산을 종료한다. (즉, *operand* 가 2 일때 까지만 곱셈을 한다)

3. System overview

Figure 1 은 구현할 시스템의 블록 다이어그램이다. 본 시스템은 Factorial core, BUS, Memory 로 구성되고 testbench 를 이용하여 시스템의 동작을 제어한다. Factorial core 는 BUS 를 통해 접근할 수 있는 register 집합 (*opstart, opclear, opdone, intrEn, operand, result_h, result_l*)을 가지고 있다. Factorial core 는 이 register 집합을 통해 외부 모듈과 데이터를 주고받을 수 있다. Factorial core 의 각 register 별 기능과 register address map 은 뒤의 자세한 내용을 참고한다.

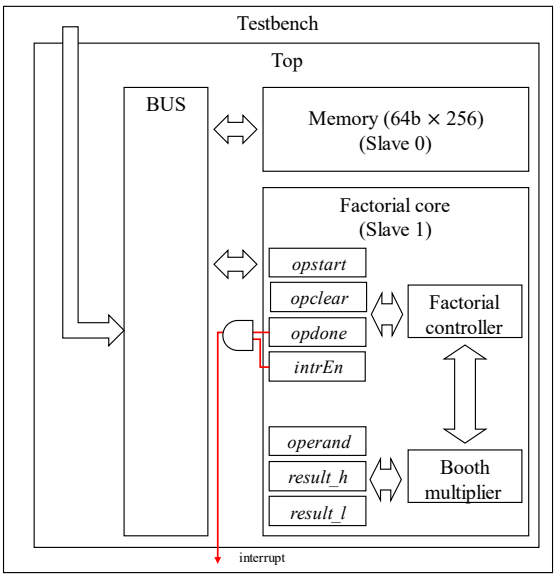

Figure 1. System overview

본 시스템이 시작하면 testbench 는 BUS 를 통해 Factorial core 에 접근하여 register read/write 를 수행한다. 예를 들어 연산을 시작하기 위해서는 testbench 가 *operand* 에 피연산자를, *opstart[0]*에 1 을 순서대로 write 해야 한다. 만약 testbench 가 interrupt 사용을 원하면 *opstart[0]*에 1 을 write 하기 전 *intrEn*[0]에 1 을 write 해야 한다. 연산 종료 후에는 *opclear[0]*에 1 을 write 하여 Factorial core 를 초기화한다.

아래 내용은 시스템 수행 순서를 자세히 나타낸 것이다. 각 모듈의 pin 에 대해서는 5.2, 6.2 장을, register 의 이름과 기능은 4.3 장을 참조하도록 한다.

1. Testbench 는 Factorial core 의 *operand* 에 factorial 의 피연산자를 write 한다.
   - ✓ 이때 *operand* register 에는 음수 혹은 $2^{31} - 1$ 을 초과하는 값은 들어오지 않는다고 가정한다.
2. Testbench 는 *intrEn[0]*을 통해 interrupt 사용 여부를 결정한다. Interrupt 를 사용하는 경우 1, 사용하지 않는 경우 0 을 write 한다.
   - ✓ *intrEn[0]* == 1'b1 인 경우에만 `interrupt` 가 발생한다.

3. Testbench 는 *opstart[0]*에 1 을 write 하여 Factorial core 의 연산을 시작한다.
4. Testbench 는 interrupt 사용 여부에 따라 다음 두가지 방법 중 하나로 연산 진행 상황을 파악한다.
   - ✓ Interrupt 를 사용하는 경우 – Interrupt
     - Testbench 는 `interrupt` 신호가 발생할 때까지 대기한다.
   - ✓ Interrupt 를 사용하지 않는 경우 – Polling
     - 연산이 종료될 때까지 매 사이클 *opdone* 을 읽는다. Testbench 는 core 의 동작이 완료될 때까지 *opdone* 이외의 register 에 접근하지 않는다.
     - *opdone[1:0]* == 2'b00: 연산 대기
     - *opdone[1:0]* == 2'b10: 연산 시작
     - *opdone[1:0]* == 2'b11: 연산 완료
     - *opdone[31:2]*: don't care
5. Factorial core 는 *opstart[0]*이 1 이 되면 연산을 수행한다. 연산을 시작할 때 다음과 같은 동작이 수행된다.
   - ✓ (0! 에 대한 예외) 만약 *operand* 의 값이 0 이면 곱셈을 수행하지 않고 바로 *result_h* = 64'd0, *result_l* = 64'd1, *opdone[1:0]* = 2'b11 로 만들어 연산을 종료한다.
   - ✓ *operand* 의 값과 *result_l* 의 값을 곱함으로써 (단, *result_l* == 64'd0 인 경우 *result_h* 의 값을 곱한다) factorial 연산을 수행한다. 곱셈 연산은 booth multiplier 를 통해 수행된다. 곱셈의 결과 128bits 중 상위 64bits 는 *result_h* 에, 하위 64bits 는 *result_l* 에 저장한다. 곱셈이 한번 끝날 때 마다 *operand* 의 값은 1 이 될 때까지 1 씩 감소하며, *operand* 가 1 이 되면 더 이상 곱셈을 수행하지 않고 연산을 종료한다. (즉, *operand* 가 2 일때 까지만 곱셈을 한다)
   - ✓ 연산을 진행하는 상태를 나타내기 위하여 *opdone[1]*에 1 을 write 한다.
6. Factorial core 는 *operand* register 의 값이 1 이 되면 더 이상 곱셈을 하지 않고 연산을 완료하며, 연산이 완료될 때 다음과 동작을 수행한다.
   - ✓ *opdone[0]*에 1 을 write 하여 연산 종료를 나타낸다.
   - ✓ *intrEn[0]* == 1 인 경우 *opdone[0]*과 *intrEn[0]*이 AND 연산되어 `interrupt` 가 발생한다

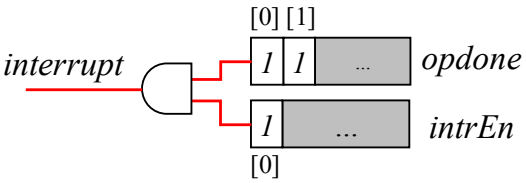

Figure 2. `interrupt` signal

7. Testbench 는 연산이 종료되면 (`interrupt` == 1'b1 혹은 *opdone[1:0]* == 2'b11)일 때 Factorial core 의 *result_l, result_h* 를 읽는다.
   - ✓ Testbench 는 bus 를 통해 한 번에 하나의 address 에서만 값을 읽을 수 있으므로 총 2 회에 걸쳐 값을 읽도록 한다.

8. Testbench 는 7 번 과정을 통해 읽은 factorial 계산 결과를 Memory 에 쓴다. 주소는 Testbench 가 Memory 의 address range 내에서 임의의 주소를 지정한다.
   - ✓ 이때 각 주소는 겹치지 않도록 하여 계산 결과가 overwrite 되지 않도록 유의한다.
9. Testbench 는 다음 연산을 수행할 수 있도록 Factorial core 를 초기화한다. 이때 초기화는 다음 방법으로 수행한다.
   - ✓ Testbench 가 Factorial core 의 *opclear[0]*에 1 을 써 Factorial core 내의 *opclear*, *intrEn, operand* 를 제외한 나머지 register 값을 초기화 한다.
   - ✓ *opclear* register 는 testbench 가 clear 이후 직접 0 을 쓴다. (즉, testbench 가 *opclear* 를 해제한다)
10. 1 에서 9 번까지의 과정을 필요에 따라 반복한다.

- ➢ 단, 위에 작성된 testbench 의 동작은 예시이므로 설계에 따라 testbench 의 데이터 저장 순서 등 Top module 의 동작에 영향을 주지 않는 부분은 본 문서와 일치하지 않더라도 무방합니다.
- ➢ 위 과정에서 memory read 등 specification 상 정의는 되어있으나 동작에 필요하지 않는 부분이 빠져 있습니다. 그러나 학생분들은 반드시 해당 부분도 정상적으로 동작하는지 검증하시기 바랍니다. 학생분의 testbench 에서는 생략되어 문제가 발생하지 않았다고 하더라도 채점 시 사용하는 testbench 에서 문제가 발생하는 경우 감점됩니다.

4. Factorial core

Factorial core 는 주어진 operand 의 값에 대해 factorial 연산을 수행하는 모듈이다.

4.1. Features

- Factorial 연산의 결과는 *result_h, result_l* 에 저장되어야 한다.
  - ✓ 각각 상위 64 bits, 하위 64 bits 를 저장한다.
- Operand 감산을 제외한 모든 산술 연산은 operator (예: 곱셈: *, 덧셈 +, 뺄셈 -, 나눗셈 /, 나머지 %)로 구현하면 안 된다. (논리 연산 등 다른 연산은 operator 를 사용해도 된다)
  - ✓ Operand 감산을 제외한 모든 덧셈 및 뺄셈은 adder 를 구현하여 이용하도록 한다.
  - ✓ 곱셈은 booth multiplier 를 이용한다. Radix 는 자유롭게 구현한다.
    - Radix 에 따른 가산점은 9 장을 참조한다.
- *opstart[0]*가 1 일 경우 *operand* 에 저장된 값을 이용해 연산을 시작한다.
  - ✓ Operand 는 32 bits signed binary number 의 양수 범위에서 주어진다고 가정한다.
  - ✓ 연산 시 *operand* register 의 값을 내부 register 에 복사하여 사용하거나 operand register 를 직접 이용할 수 있다. 어느 방법을 이용해도 최종 결과에는 영향을 주지 않는다.
- 연산을 시작할 때 *opdone[1]*에 1 을 쓴다.
- 연산이 끝날 경우 *opdone[0]*에 1 을 쓰고 대기한다.
- *intrEn[0]*이 1 일 경우 연산이 끝났을 때 `interrupt` 가 발생한다. `interrupt` 는 BUS 를 거치지 않고 Top module 을 통해 바로 testbench 와 연결된다.
- *opclear[0]* 가 1 일 경우 *opclear*, *intrEn*, *operand* 를 제외한 나머지 register 를 초기화 한다.
  - ✓ 연산 시작 전 혹은 연산 도중 *opclear[0]* 이 1 이 되는 상황은 가정하지 않는다.
  - ✓ Clear 이후 testbench 는 opclear[0]에 0 을 써 clear 동작을 마무리한다.
  - ✓ 자세한 register 의 동작과 기능은 4.3 을 참고
- Figure 3 은 Factorial core 의 schematic symbol 로 input/output port 의 이름과 bit width 를 나타낸다.

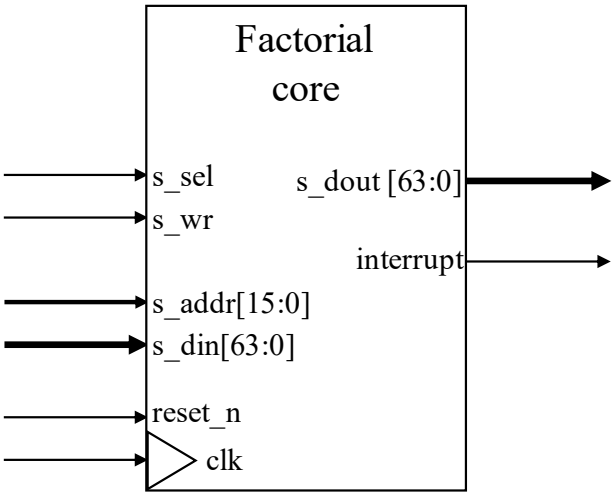

Figure 3. Schematic symbol of Factorial core

4.2. Pin description

- Module 명은 `FactoCore` 이다.
- Table 1 는 `FactoCore` 의 port 의 이름과 방향을 설명한다.
- Port 의 이름과 bit width 는 반드시 동일해야 한다.

Table 1. Pin description of `FactoCore`

Direction	Port name	Bit width	Description
Input	`clk`	1	Clock
	`reset_n`	1	Active low reset
	`s_sel`	1	(Slave interface) Select
	`s_wr`	1	(Slave interface) Write/read
	`s_addr`	16	(Slave interface) Address
	`s_din`	64	(Slave interface) Data input
Output	`s_dout`	64	(Slave interface) Data output
	`interrupt`	1	Interrupt out

4.3. Register description

- Table 2 는 Factorial core 내부의 register 를 설명한 것이다. Register 의 bit width 는 기본적으로 64 bits 이다.
  - ✓ Register type 은 W(write)와 R(read)로 구별된다. W(write)의 경우, slave interface 를 통해 외부에서 해당 register 에 값을 write 할 수 있다. R(read)의 경우, slave interface 를 통해 외부에서 해당 register 의 값을 read 할 수 있다.
    - Testbench 는 W type register 을 읽지 않으며 R type register 에 값을 쓰지 않는다.
  - ✓ Register 에서 사용되지 않는 bit 는 reserved 이다. 예를 들어 '*opstart[63:1]*가 reserved 이다'의 의미는 *opstart* register 의 [63:1] bits 에 쓰여진 값은 무시되며, read 된 값은 의미가 없음을 나타낸다.
  - ✓ Default value 는 reset 이 되었을 때 초기 값을 의미한다. 0 이 아닌 경우가 있으므로 반드시 확인하도록 한다.

Table 2. Register description of `FactoCore`

Offset	Type	Bit width	Name	Description	Default value
0x00	W	64 bits	*opstart*	[0] bit 에 1 이 써지면 *operand* register 의 값을 이용해 연산을 시작한다. 연산 중 혹은 연산 완료 후 register 초기화 이전에 해당 register 의 [0]에 1 이 써지면 해당 값은 무시된다. [63:1] bits 는 reserved 이다.	64'd0
0x08	W	64 bits	*opclear*	해당 register 의 [0] bit 에 1 이 써지면 *opclear*, *intrEn*, *operand* 를 제외한 나머지 register 의 값이 default value 가 된다. *opclear* 를 통한 reset 은 clock 과 synchronous 하다. [63:1] bits 는 reserved 이다.	64'd0
0x10	R	64 bits	*opdone*	연산이 시작될 경우 [1] bit 에 1 을 써 동작을 수행하고 있음을 나타낸다. Core 의 연산이 완료되었을 경우 [0] bit 에 1 을 써 연산이 끝났음을 기록한다. [63:2] bits 는 reserved 이다.	64'd0
0x18	W	64 bits	*intrEn*	연산이 종료되었을 때 interrupt 신호를 발생시키기 위해 사용한다. intrEn[0] ==	64'd0

				1 인 경우에만 interrupt 가 발생한다. [63:1] bits 는 reserved 이다.	
0x20	W	64 bits	*operand*	Factorial 연산의 피연산자를 나타낸다. 예를 들어 7!을 수행하고 싶은 경우 operand 에 7 을 저장한다. 즉, 32 bits signed number 를 저장한다. [63:32] bits 는 reserved 이다.	64'd0
0x28	R	64 bits	*result_h*	Factorial 연산 결과를 저장한다. 곱셈 결과의 상위 64bits 를 저장한다.	64'd0
0x30	R	64 bits	*result_l*	Factorial 연산 결과를 저장한다. 곱셈 결과의 하위 64bits 를 저장한다.	64'd1

4.4. Functional description

Factorial core 는 BUS 와 연결된 slave component 이다. BUS 와 연결된 master (testbench)는 Factorial core 의 slave interface (`s_sel, s_wr, s_addr, s_din`)를 통하여 Factorial core 의 내부 동작을 제어한다. Master 는 slave interface 를 통하여 Factorial core 에 포함된 register 에 미리 정의된 값을 써 제어한다. 예를 들어, *opstart[0]*에 1 을 써넣어 연산을 시작할 수 있다. 각각의 register 들은 offset 이 할당되어 있으며 register 의 bit 위치마다의 값에 따라서 모듈의 동작을 제어한다.

Offset 은 기준 주소(base address)와 목표 주소(target address)의 차이를 지칭할 때 쓰는 단어이다. Factorial core 의 기준 주소가 0x7000 이며 Factorial core 의 *result_l* register 주소가 0x7030 이다. 그러면 offset 은 0x0030 이 된다. 여기서 offset 의 하위 3bits 는 byte addressing 을 위한 주소로, 하위 3bits 는 어떤 값이 들어오더라도 데이터 시작 위치에는 영향을 주지 않는다. 즉, 0x7035 에 접근하더라도 0x7030 에 접근한 것과 동일하게 *result_l* 을 읽을 수 있다. 만약, Factorial core 의 register offset 을 넘어가는 주소가 입력될 경우 Factorial core 는 해당 입력을 무시한다.

Factorial core 의 slave interface 는 `s_addr` 를 이용해 offset 을 계산한다. Offset 과 `s_wr` 를 이용해 register 에 `s_din` 데이터를 write 하거나 register 값을 `s_dout` 으로 내보낸다. `s_wr` 이 0 일 경우 register read 를, `s_wr` 이 `1` 일 경우 register write 를 한다.

Factorial core 의 *opstart*[0]에 1 이 write 될 경우 동작 순서는 다음과 같다

1. 모듈이 작동중임을 표시하기 위해 *opdone[1]*에 1 을 쓴다.
2. *operand* 의 값을 읽는다. *operand* 의 값이 $2^{31} - 1$ 을 넘어가는 경우는 없다 가정한다.
3. *operand* 의 값에 따라 factorial 연산을 수행한다.
   - ✓ (0! 에 대한 예외) 만약 *operand* 의 값이 0 이면 곱셈을 수행하지 않고 바로 *result_h* = 64'd0, *result_l* = 64'd1, *opdone*[1:0] = 2'b11 로 만들어 연산을 종료한다.
   - ✓ *operand* register 의 값과 *result_l* 의 값을 곱함으로써 (단, *result_l* == 64'd0 인 경우 *result_h* 의 값을 곱한다) factorial 연산을 수행한다. 곱셈 연산은 booth multiplier 를 통해 수행된다. 곱셈의 결과 128bits 중 상위 64bits 는 *result_h* 에, 하위 64bits 는 *result_l* 에 저장한다. 곱셈이 한번 끝날 때 마다 *operand* register 의 값은 1 이 될 때까지 1 씩 감소한다.
   - ✓ *operand* register 의 값이 1 이 되면 더 이상 곱셈을 수행하지 않는다. 즉, *operand* 가 1 인 경우에는 *result_h* 와 *result_l* 의 값이 더 이상 바뀌지 않는다.
4. 앞선 과정을 통해 factorial 연산이 완료되었을 때 *opdone[0]*에 1 을 쓴다.
   - ✓ *intrEn[0]* == 1'b1 이라면 이와 동시에 interrupt 가 발생한다.
5. 아래의 방법으로 Factorial core 가 초기화 되기 전까지 register 값을 유지한다. 초기화 전까지 master device 는 새로운 연산을 시작하도록 명령을 줄 수 없다.
   - ✓ *opclear[0]* 가 1 이 될 경우 *opclear, intrEn*, *operand* 를 제외한 나머지 register 값을 default value 로 초기화한다. Clear 이후에는 testbench 가 *opclear[0]*에 0 을 쓴다.

5. BUS

BUS 는 여러 component 들 간에 data 를 전송(transfer)할 수 있도록 연결해주는 component 이다. BUS 는 새로운 component 들을 추가하기가 쉬우며, 가격이 저렴한 특징을 가지고 있다.

5.1. Features

- BUS 는 1 개의 master interface 와 2 개의 slave interface 를 가지고 있다.
- Address 는 16 bits 이다.
- Data 는 64 bits 이다.
- Slave 0 (Memory)은 0x0000 ~ 0x07FF 사이의 address 를 memory map region 으로 가진다.
- Slave 1 (Factorial core)은 0x7000 ~ 0x71FF 사이의 address 를 memory map region 으로 가진다.
- Master device 가 slave 0 과 slave1 address area 이외의 주소에 접근하거나 `m_req` 가 0 인 경우 `s0_sel, s1_sel` 를 0 으로 바꾼다.
  - ✓ 즉, 어떤 slave device 도 선택되지 않는다.
- Figure 4 은 BUS 의 schematic symbol 로 input/output port 의 이름과 bit width 를 나타낸다.
- 왼쪽은 master interface 를 나타내며, 오른쪽은 slave interface 를 나타낸다.

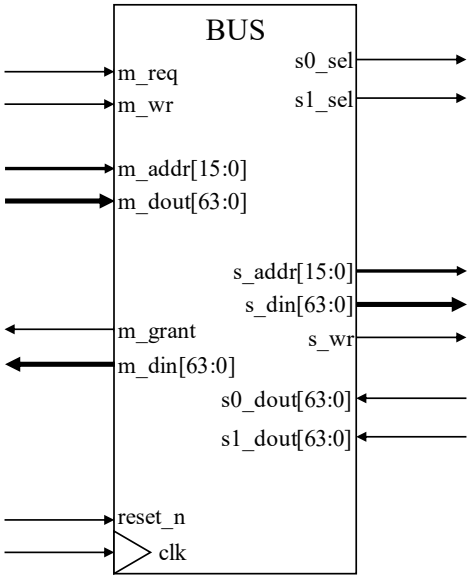

Figure 4. Schematic symbol of bus

5.2. Pin description

- Module 명은 `BUS` 이다.
- Table 3 은 `BUS` 의 pin 을 정리한 것이다.
- Port 의 이름과 bit width 는 반드시 동일해야 한다.
  - ✓ ***Notice***: Master 에서 data-out 이 output pin 이지만, BUS 에서는 해당 data-out 을 받아야 하기 때문에 data-out 이 input pin 이 된다.
  - ✓ ***Notice***: master 에서 data-in 이 input pin 이지만, BUS 에서는 data-in 이 output pin 이 된다. 이는 slave 에도 똑같이 적용된다.

Table 3. Pin description of `BUS`

Direction	Port name	Bit width	Description
Input	`clk`	1	Clock
	`reset_n`	1	Active low reset
	`m_req`	1	Master request
	`m_wr`	1	Master write/read
	`m_addr`	16	Master address
	`m_dout`	64	Master data output
	`s0_dout`	64	Slave 0 data out
	`s1_dout`	64	Slave 1 data out
Output	`m_grant`	1	Master grant
	`m_din`	64	Master data input
	`s0_sel`	1	Slave 0 select
	`s1_sel`	1	Slave 1 select
	`s_addr`	16	Slave address
	`s_wr`	1	Slave write/read
	`s_din`	64	Slave data input

5.3. Functional description

프로젝트에서 구현할 `BUS` 는 1 개의 master interface (Testbench)와 2 개의 slave interface (Factorial core, memory)로 구현되어 있다.

아래의 내용은 `BUS` 의 동작 순서이다. 이번 프로젝트는 master 가 1 개이므로 `BUS` 를 사용할 때 여러 개의 master 가 bus 를 요청하는 동작은 생략되어 있다.

1. Master 는 `BUS` 를 통해 slave 와 communication 하고자 할 때, request signal 인 `m_req` 신호를 1 로 바꿔 BUS 사용을 가능 여부를 확인한다.

2. `BUS` 는 master 가 bus 사용을 요청했을 때 master 에게 grant signal 인 `m_grant` 신호를 통해 `BUS` 사용을 허락한다. 다시 말해, `m_req` 는 master 가 `BUS` 에 대한 소유권을 요청하는 것이며, `m_grant` 신호는 `BUS` 가 master 에게 `BUS` 에 대한 소유권을 할당해주는 것이다.

3. Master 가 `m_grant` 신호를 받은 후에는 `m_addr` 신호를 통해 slave 의 memory map 에 따라 slave 와 communication 을 수행한다.

4. Master 의 communication 동작이 완료될 경우 `m_req` 신호를 0 으로 만들어 BUS 사용을 끝내며 BUS 는 `m_grant` 신호를 0 으로 만든다. 만약, Master 의 `m_req` 신호가 1 인 동안에는 `m_grant` 가 0 이 되지 않으며 communication 를 계속할 수 있다.

6. Memory (RAM)

RAM(random access memory)은 임의의 address 에 대해 data 를 읽고 쓰는 memory 를 의미한다. 이 프로젝트에서 지칭하는 memory 는 모두 RAM 을 의미한다.

6.1. Features

- Address 는 8 bits 이다.
- Memory 의 address 는 `BUS` 의 LSB(least significant bit)부터 address 를 받는다.
  - ✓ Memory 의 `s_addr` 의 LSB 는 `BUS` 의 `s_addr` 의 4 번째 비트와 일치한다. 다시 말해 Memory 의 `s_addr` 의 MSB 는 `BUS` 의 `s_addr` 의 8 번째 bit (`s_addr[10]`)와 일치한다. 정리하면, memory 의 `s_addr` 은 `BUS` 의 `s_addr[10:3]`과 같다.
- Data 는 64 bits 이다.
- Memory 는 내부에 256 개의 data 를 address 에 기반하여 저장한다.
- Figure 5 은 Memory 의 schematic symbol 로 input/output port 의 이름과 bit width 를 나타낸다.

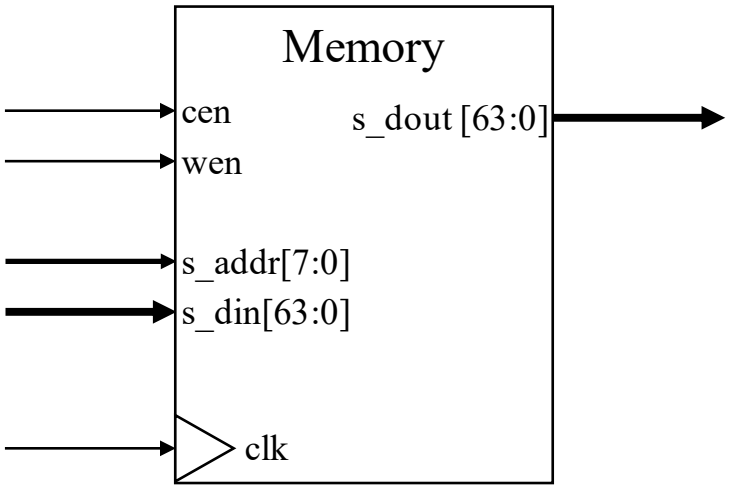

Figure 5. Schematic symbol of memory

6.2. Pin description

- Module 명은 `ram` 이다.
- Table 4 은 `ram` 의 pin 을 정리한 것이다.
- Port 의 이름과 bit width 는 반드시 동일해야 한다.

Table 4. Pin description of `ram`

Direction	Port name	Bit width	Description
Input	`clk`	1	Clock
	`cen`	1	Chip enable
	`wen`	1	Write enable
	`s_addr`	8	Address
	`s_din`	64	Data in
Output	`s_dout`	64	Data out

6.3. Functional description

Memory 모듈의 동작은 아래와 같다

- `cen` 과 `wen` 이 모두 1 이면 address (`s_addr`) 가 가리키는 memory 에 `s_din` 을 write 한다. 이때 `s_dout` 은 0 을 출력한다.
- `cen` 이 1 이고, wen 이 0 이면, address (`s_addr`) 가 가리키는 memory 의 값을 `s_dout` 에 write 한다. `cen` 이 0 이면, `s_dout` 은 0 이 된다.

7. Top

Top 모듈은 BUS, Factorial core 와 Memory 를 instance 하여 연결한 모듈이다. Top 모듈의 input port 를 이용해 Top 모듈 내에 있는 BUS 의 master port 에 접근이 가능하다. Top 은 memory mapped I/O 방식을 사용하여 외부(testbench)에서 주소를 통해 Top 모듈의 slave (Factorial core, Memory) device 에 접근이 가능하다. 또한 Top 은 내부의 Factorial core 에 직결된 `interrupt` 를 통해 core 의 동작 상태를 testbench 에 전달할 수 있다.

7.1. Features

Figure 6.는 Top 모듈의 schematic symbol 이다. BUS 의 master interface 가 input/output port 를 통해 연결되어 있다. 즉, BUS 의 master pin 들은 Top module 을 통해 testbench 가 접근하게 된다. Factorial core 의 `interrupt` pin 이 Top 의 `interrupt` port 와 바로 연결되어 있다. Top 모듈 내의 memory map 은 Table 5 과 같다.

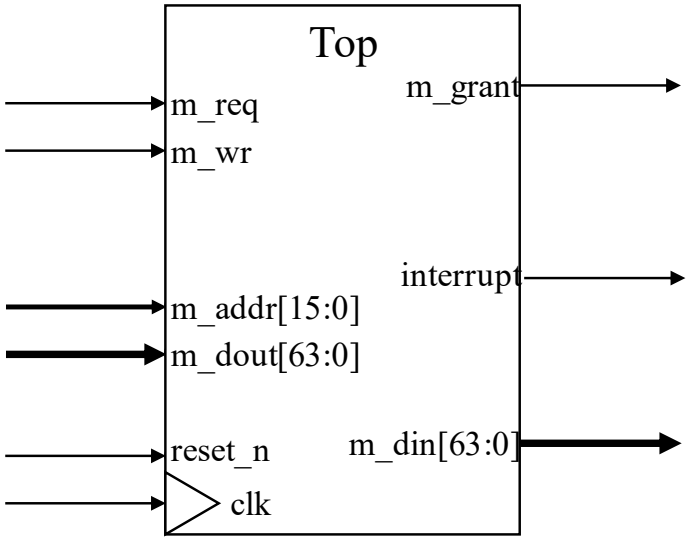

Figure 6. Schematic symbol of Top

Address region	Component
0x0000 - 0x07FF (0b 0000 0000 0000 0000 ~ 0b 0000 0111 1111 1111)	Memory
0x7000 - 0x71FF (0b 0111 0000 0000 0000 ~ 0b 0111 0001 1111 1111)	Factorial core

Table 5. Memory map

7.2. Pin description

- Module 명은 `Top` 이다.
- Table 6. Pin description of `Top` 는 `Top` 의 port 의 이름과 방향을 설명한다.
- `Top` 의 port 들은 하위 module 들의 pin 을 그대로 port 로 연결한 것이다.
  - ✓ `Top` 의 m_* port 들은 BUS 의 master interface port 와 이름 및 기능이 동일하다. 동작에 관해서는 5.3 을 참조하도록 한다.
  - ✓ `Top` 의 interrupt port 는 Factorial core 의 interrupt port 와 이름 및 기능이 동일하다. 동작에 관해서는 4.4 를 참조하도록 한다.
- Port 의 이름과 bit width 는 반드시 동일해야 한다.

Table 6. Pin description of `Top`

Direction	Port name	Bit width	Description
Input	clk	1	Clock
	reset_n	1	Active low reset
	m_req	1	Master request
	m_wr	1	Master write/read
	m_addr	16	Master address
	m_dout	64	Master data output
Output	m_grant	1	Master grant
	m_din	64	Master data in
	interrupt	1	Factorial core interrupt

8. Report outline

- 제안서
  - ✓ 발표: PowerPoint 로 발표 희망자에 한해 10 분 분량으로 작성
  - ✓ 보고서: 최소한 다음의 내용들이 포함되어야 하며 그 외의 것을 추가하는 것은 자유다.
    - 과제제목
    - 과제목표
    - 일정
    - 각 모듈 별 구현 방법 (state transition diagram 및 내부 register map 등) 기술
    - 예상되는 문제점
    - 검증전략
      - 구현 방법이 아님에 유의 (구현 방법 작성시 감점)
- 결과보고
  - ✓ 보고서: 최소한 다음의 내용들이 포함되어야 하며 그 외의 것을 추가하는 것은 자유다.
    - Introduction
    - 일정 및 계획을 포함할 것
    - Project specification
    - Design details
    - Design verification strategy and results
    - Conclusion
      - 과제 완료 후 기대되는 학습효과를 포함할 것.
  - ✓ 보고서의 ideal 한 양식은 논문의 형태입니다. (공지사항에 양식 업로드 예정)

9. Grading

- Scoring criteria
  - ✓ 제안서: 20%
  - ✓ Testbench: 40%
    - BUS: 15%
    - Memory: 15%
    - Factorial core: 30%
    - Top: 40%
  - ✓ 결과보고서: 40%
  - ✓ Testbench 이용 결과 검증
    - 20 개의 testbench 를 이용하여 결과 검증
    - Testbench 내용은 미공개
- Reason of score deduction
  - ✓ 파일 이름에 한글이 포함된 경우 감점
  - ✓ Verilog code 파일 (.v) 내용에 한글이 포함된 경우 감점
  - ✓ 제출 형식(파일명 등)을 지키지 않을 경우 감점
  - ✓ Compile 불가시 사유에 따라 차등적으로 감점
  - ✓ 상기 사유 이외에도 교수/조교 판단 하에 필요하다면 감점이 이루어질 수 있음
- Design and implementation bonus
  - ✓ Faster booth multiplication
    - Radix-4 : +1
    - Radix-16 : +2
    - 이외 더 큰 Radix 를 사용해도 무방하나 가산점은 +2 점까지만 부여함
  - ✓ Smallest area bonus
    - 같은 radix 의 booth multiplier 로 구현한 학생들의 결과물과 비교하여 각 radix 별 area 낮은 순으로 상위 k 명에게 보너스 점수를 차등적으로 부여 (5 점, 4 점, …, 1 점)
    - $k = min(5, ceil(\text{같은 } radix\text{로 제출한 인원} / 20))$
  - ✓ Highest throughput bonus
    - Throughput = Maximum clock frequency * Output per clock cycle
    - 가장 throughput 이 높은 상위 5 명에게 보너스 점수를 차등적으로 부여 (5 점, 4 점, …, 1 점)
    - Throughput bonus 를 받기 위해서는 결과 보고서에 최소 다음 두 사항을 작성해야 함
      - Time Quest Timing Analyzer 의 Maximum frequency report
      - 본인이 구현한 Factorial core 의 clock cycle 당 output 데이터 개수와 그 이유
    - Throughput 의 기준이 되는 data 는 차후 KLAS 공지사항에 업로드 예정
    - Throughput bonus 를 받기를 원하지 않는다면 위 사항을 작성하지 않아도 괜찮으나 허위로 작성하는 경우 프로젝트 점수에서 5 점을 감점함

- Presentation bonus
  - ✓ 제안서 발표 (완료)
    - 발표 희망자는 조교에게 메일 보낼 것 (선착순 3 명)
    - 이메일: kdsh5800+dlc2_project@gmail.com
    - 신청 기한: 2023 년 11 월 12 일까지
    - 발표일: 2023 년 11 월 16 일 목요일 수업시간 중

[주의] 발표하기로 한 학생이 특별한 사유 없이 불참한 경우 전체 점수 감점 등의 불이익 발생할 수 있음. 만약 완성하지 못했더라도 가능한한 자료를 만들어 발표를 진행하시기 바랍니다.

10. Submission

- Issue date: 2023 년 11 월 07 일 화요일 (10 주차 화요일)
- 과제 제안서 제출 (완료)
  - ✓ Due date: 2023 년 11 월 16 일 목요일 23:59 (11 주차 수업 전까지 제출)
  - ✓ Late submission: 2023 년 11 월 17 일 금요일 23:59
  - ✓ 컴퓨터공학기초실험 2 수강자는 실습 수업의 과제 제출 란에 제출
  - ✓ 미수강자는 디지털논리회로 2 과목의 과제 제출 란에 제출
  - ✓ 파일명: Proposal_(분반)_(학번).pdf
- 최종 결과 제출
  - ✓ Due date: 2023 년 12 월 06 일 수요일 23:59 (15 주차 수업 전일까지 제출)
  - ✓ **늦은 제출은 받지 않습니다.**
  - ✓ 컴퓨터공학기초실험 2 수강자는 실습 수업의 과제 제출 란에 제출
  - ✓ 미수강자는 디지털논리회로 2 과목의 과제 제출 란에 제출
  - ✓ 최종 결과 제출시 source code 를 보고서와 같이 압축하여 업로드한다
    - Source code 는 구현에 사용된 모든 Verilog 파일 및 이에 대한 testbench 를 의미한다
      - 즉, Verilog (.v) 파일과 보고서 PDF 를 압축하여 제출하면 된다.
      - Quartus project 파일은 필요하다면 (e.g., compile 옵션 변경) 제출한다.
    - 최소한 다음 모듈에 대한 testbench 가 source code 에 포함되어야 한다
      - Top
      - Bus
      - Memory
      - Factorial core
  - ✓ 파일 압축할 때 db, incremental_db, simulation 폴더는 제외한다.
  - ✓ 파일명: Project_(분반)_(학번).zip
- 유의사항
  - ✓ 분반은 본인의 컴퓨터공학기초실험 수강 분반에 따라 작성한다.
    - 미수강: 0, 월: 1 , 수: 2, 금: 3

      예) 컴퓨터공학기초실험 월요일 분반: Project_1_2022112233.zip
  - ✓ 파일명 뒤에 버전을 기록해 KLAS 에 여러 번 제출해도 된다.
    - 예) Project_(분반)_(학번)_v2.zip
  - ✓ 채점의 경우 최종 버전을 기준으로 한다. 같은 버전의 파일이 두개 이상 업로드 된 경우 마지막 제출 파일만 반영한다.
  - ✓ 잘못된 파일이 업로드 되거나 파일이 깨진 경우 점수에 불이익이 발생할 수 있으니 반드시 제출 후 KLAS 에서 업로드한 파일을 다시 다운로드 받아 문제가 없는지 확인하도록 한다.