한국기술교육대학교

Embedded SoC

범용 병렬 연산기 설계

Term project - Designing General Purpose Parallel Processing Unit

2014161001 강승우

2019-6-21

개요

Graphics Processing이나, Digital Signal Processing 등의 작업을 하게 되면 일반적으로 절차 처리에 특화된 CPU로는 감당하기 어려운 수준의 데이터를 다루게 된다.

그러나 이 과정에서 다루게 되는 대부분의 연산은 대체로 행렬 연산으로 이루어져 Deterministic하고 반복적이므로, CPU의 처리 능력을 낭비하는 대신 간단한 연산을 대량으로 처리해줄 디바이스가 고안되었다.

본 자유수행 과제의 목표는 초보적인 수준의 병렬 연산기를 구현하는 것으로, Intel-EP4CE 칩셋과, 후면에 탑재한 Terasic 사(社)의 VEEK-MT2 FPGA 개발 키트가 사용되었다.

목적

범용 병렬 연산기는 다음의 기능을 포함하게 된다.

- 기본적인 논리 연산 / 산술 연산
- 24개의 프로세싱 엘리먼트 (스레드로 표현)
- 48Kbyte의 정적 메모리
- 멀티 사이클 FPU
- 명령어 수준 병렬성 구현

기본적인 논리 연산 / 산술 연산

범용 병렬 연산기는 덧셈, 뺄셈, 논리 연산, 1비트 단위 시프트 연산을 지원하는 ALU를 내장한다. ALU는 기본적인 상태, VCNZ를 반환한다.

24개의 프로세싱 엘리먼트

범용 병렬 가산기는 기본적으로, 하나의 명령어를 다수의 프로세서가 동시에 실행하는 SIMD 개념을 구현하고 있다.

하나의 코어 로직이 명령어를 Fetch, Decoding해 컨트롤 워드를 플랫폼에 올리면, 그것은 모든 프로세싱 엘리먼트가 공유하여 같은 명령을 수행하게 된다.

48Kbyte 정적 메모리

각각의 프로세싱 엘리먼트 내부에는 2Kbyte 크기의 정적 메모리가 들어있다. 메모리는 버스가 아닌, 멀티플랙싱을 통해 외부 단자와 직결되며, 이를 통해 손쉽게 데이터를 교환할 수 있다.

멀티 사이클 FPU

FPU의 설계는 그것만 갖고도 하나의 과제가 될 만큼 방대한 주제이므로, 처음부터 만들기보단 기존의 IP를 십분 활용하였다.

인텔의 NIOS 프로세서 확장 명령어용 멀티 사이클 FPU를 인터페이스를 맞춰서 각각의 프로세싱 엘리먼트에 공급하며, 아래와 같은 명령어를 제공한다.

- Int to float
- Float to int
- Sqrt
- Add
- Sub
- Multiply
- Divide

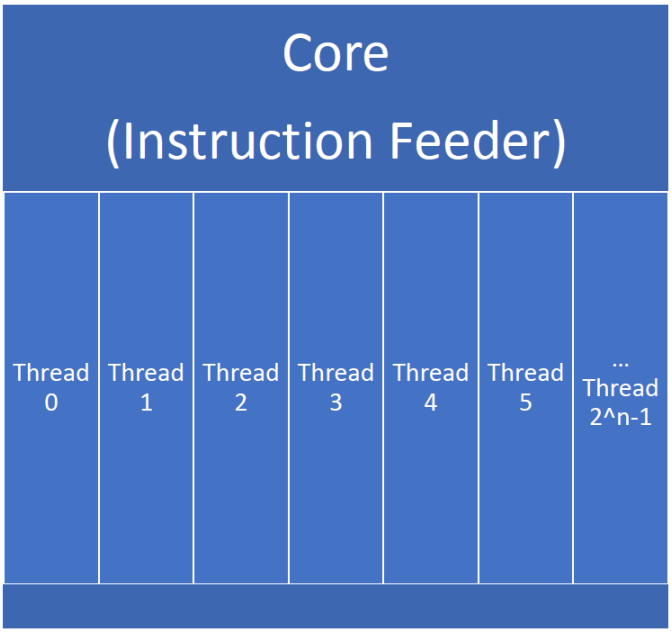
최대한의 퍼포먼스를 내기 위해 파이프라인 구조를 사용하는데, 멀티 사이클 FPU가 동작하는 동안에는 파이프라인 스테이지 전체에 stall이 걸리는 방법으로 FPU를 병합하였다.

명령어 수준 병렬성 구현

파이프라인 CPU는 클럭 당 명령어 개수를 극적으로 끌어올려주지만, 동시에 데이터 오염의 위험에서 자유로울 수 없다.

데이터가 충돌하는 지점에서 Bubble을 삽입해 파이프라인 전체를 지연시키는 방법도 있으나, 똑같은 작업을 대용량의 데이터를 대상으로 하게 되는 병렬 연산기에서는 다소 비효율적이다.

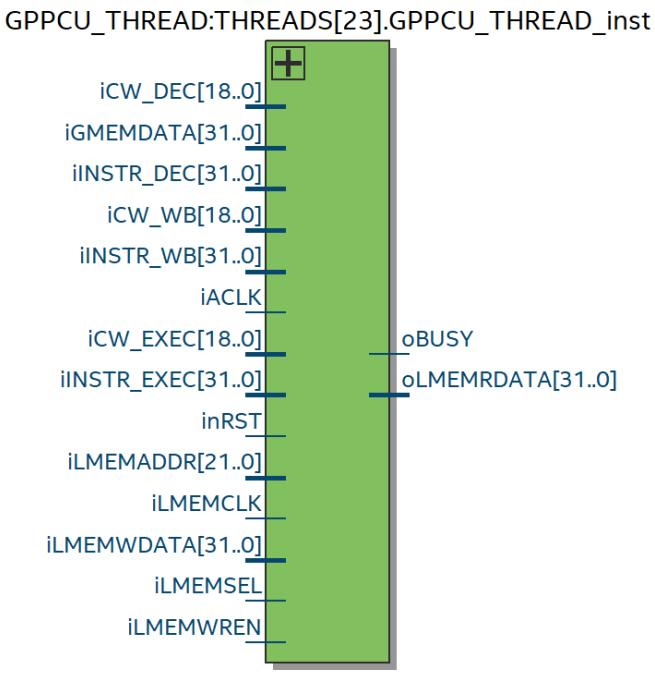
따라서, 본 병렬 연산기는 각각의 스래드에 분배된 작업을 순차적으로 수행하는 대신, 레지스터 파일을 쪼개 명령어를 뒤섞는다. 이에 대해서는 뒤에서 자세하게 서술하도록 하겠다.



구현 – 코어

병렬 연산기는 한 개의 Core 로직에 다수의 스레드가 종속된 형태를 취하고 있다.

Core는 Instruction을 Fetch, Decode하여 파이프라인 플랫폼 레지스터에 로드한다. 파이프라인 스테이지는 Fetch, Decode, Execute, Writeback의 네 단계이며, 디코딩 된 제어 워드는 Core에서 관리된다.



이렇게 Core에서 관리되는 하나의 컨트롤 워드를, 십수 개의 스레드가 동시에 전달받게 된다. 위의 블록 심볼을 보면 각각의 파이프라인 단계에 대해 명령어와 컨트롤 워드를 입력으로 받고 있는 것을 알 수 있다.

코어는 아래와 같은 인터페이스를 갖는다.

```
module GPPCU_CORE #
(
    parameter
    NUM_THREAD = 16,
    WORD_BW  = 10
)
(
    iACLK               ,        //
Instruction should be valid on ACLK's
rising edge.
    inRST               ,
    oIDLING             ,        // Doing
nothing.
    iINSTR              ,
    iINSTR_VALID        ,
    oINSTR_READY        ,   // Replaces
instruction address.

    iLMEM_CLK           ,
    iLMEM_THREAD_SEL    , // Thread
selection input
    iLMEM_ADDR          ,
    iLMEM_WDATA         ,
    oLMEM_RDATA         ,
    iLMEM_RD            ,
    iLMEM_WR            ,

    oGMEM_ADDR          ,
    iGMEM_WDATA         ,
    oGMEM_CLK
);
```

내부 파이프라인 상태 등에 의해 명령어를 Fetch 할 수 없는 상태인 경우, oINSTR_READY 신호가 비활성화된다.

명령어를 수신할 수 있을 때는 oINSTR_READY 신호가 Set 되며, 외부에서 명령어를 iINSTR 포트에 인가한 뒤 iINSTR_VALID 를 Set 시키면 다음 클럭에서 인스트럭션이 Fetch 된다.

```
always @(posedge iACLK) begin
    cw_valid_writeback  <= ~inRST ? 0 : data_stall_exec   ? 0              : cw_valid_exec;
    cw_valid_exec       <= ~inRST ? 0 : data_stall_exec   ? cw_valid_exec   : data_stall_decode  ? 0 : cw_valid_decode;
    cw_valid_decode     <= ~inRST ? 0 : data_stall_decode ? cw_valid_decode : cw_valid_fetch;
    cw_valid_fetch      <= ~inRST ? 0 : data_stall_decode ? cw_valid_fetch  : can_fetch          & iINSTR_VALID;

    cw_writeback        <= cw_exec;
    cw_exec             <= data_stall_decode ? cw_exec         : cw_decode;      You, 3 days ago • Fix core platform reg
    cw_decode           <= data_stall_decode ? cw_decode       : decoding_cw;
    // There is no cw_instr

    instr_writeback     <= instr_exec;
    instr_exec          <= data_stall_decode ? instr_exec      : instr_decode;
    instr_decode        <= data_stall_decode ? instr_decode    : instr_fetch;
    instr_fetch         <= data_stall_decode ? instr_fetch     : iINSTR;
end
```

Figure 1. 파이프라인 플랫폼 레지스터 구성

명령어를 송신하는 측에서도, oINSTR_READY 신호와 iINSTR_VALID 신호가 동시에 활성화되어 명령어가 전달되었을 때, 다음 명령어로 갱신하는 로직을 구성할 필요가 있다.

그 아래, iLMEM 로 시작하는 포트들은 모두 각 스레드 내에 위치한 메모리에 접근하기 위한 신호이다. 이 신호들은 Latch 되지 않고, 조합 회로로 멀티플랙싱 되어 스레드 내부 메모리에 직접 인가된다.

```
 wire[DBW-1:0]
lmem_out_data_mux[NUM_THREAD-1:0];
assign oLMEM_RDATA =
lmem_out_data_mux[iLMEM_THREAD_SEL];
...
.iLMEMADDR      (iLMEM_ADDR),
.oLMEMRDATA
    (lmem_out_data_mux[idx_thread]),
```

전달된 어드레스에 대해, 모든 스레드는 각자 메모리의 출력을 반환하며, 최종적으로는 스레드 셀렉션 입력에 따라 멀티플랙싱 된 하나의 출력만 외부로 전달된다.

데이터의 입력은 Write Enable 신호를 통해 제어되며, 이 신호는 스레드 셀렉션 입력과 스레드 인덱스를 비교하는 것으로 결정된다.

```
wire lmem_thread_sel
= iLMEM_THREAD_SEL == idx_thread;
...
.iLMEMSEL       (lmem_thread_sel),
.iLMEMWREN      (iLMEM_WR),
...
.iPA_WR     (iLMEMWREN & iLMEMSEL),
```

스레드 선택 입력과 WR 신호가 동시에 인에이블 되었을 때 해당 스레드에 입력 데이터가 쓰이게 된다. 이러한 인터페이스를 통해, 모듈 전체가 하나의 커다란 메모리처럼 보일 수 있다.

GMEM 은 모든 스레드에서 같은 시점에 공통으로 사용해야 하는 상수(1, PI, E, 물체 행렬 등 ...)를 관리하는 로직이다. 따라서, 외부로 GMEM 에 대한 데이터를 요청하는 어드레스는 스레드가 아닌, 코어에서 유일하게 생성된다.

```
assign oGMEM_CLK  = ~iACLK;
assign oGMEM_ADDR =
instr_decode[INSTR_IMM2_17+:17];
```

또한, GMEM의 경우 어드레스를 디코딩 시점에 출력하고 바로 다음 스테이지에서 데이터를 이용해야 하기 때문에, 클럭 신호를 부논리로 변환하여 반 클럭 일찍 데이터를 요청, Execution Stage에 바로 로드할 수 있게끔 한다.

Thread 내부에는 단일 사이클로 실행되는 ALU와 더불어, 멀티 사이클로 실행되는 FPU가 같이 내장되어 있다. 멀티 사이클 작업이 실행되는 동안에는 Execution 스테이지를 포함한 이전의 모든 스테이지가 Stall되어야 하기 때문에, 파이프라인에 관련 로직을 구성하였다.

추가로, 데이터 해저드를 방지하기 위한 Stall Generation 로직도 추가해 넣었으나, 올바르게 동작하지 않아 결국 소프트웨어로 처리하게 되었다.

Fetch 단계의 인스트럭션을 디코딩하는 디코더이다. 조합 회로로 동작한다.

```
GPPCU_INSTR_DEC GPPCU_INSTR_DEC_inst(
    .iOPC(instr_fetch[INSTR_OPR_5+:5]),
    .oCW(decoding_cw)
```

```
always @(iOPC) begin /*
                        U U     L L
                        S S     M M F    A    B  G
                        E E     E E P    L    S  M R
                        R R F A M M O    O    E  E E
                        E E P L I I P    P    L  M G
                        G G O O R W C    C       R W
                        A B P P D R 3..0 3..0 1. D R
    NOP   : oCW <= 'b__0_0_0_0_0_0_0000_0000_xx_0_0;
    MOV   : oCW <= 'b__0_1_0_1_0_0_xxxx_0001_00_0_1;
    MVN   : oCW <= 'b__0_1_0_1_0_0_xxxx_0010_00_0_1;
    ADC   : oCW <= 'b__1_1_0_1_0_0_xxxx_0011_00_0_1;
    SBC   : oCW <= 'b__1_1_0_1_0_0_xxxx_0100_00_0_1;
    AND   : oCW <= 'b__1_1_0_1_0_0_xxxx_0101_00_0_1;
    ORR   : oCW <= 'b__1_1_0_1_0_0_xxxx_0110_00_0_1;
    XOR   : oCW <= 'b__1_1_0_1_0_0_xxxx_0111_00_0_1;
    ADI   : oCW <= 'b__1_0_0_1_0_0_xxxx_1000_01_0_1;
    SBI   : oCW <= 'b__1_0_0_1_0_0_xxxx_1001_01_0_1;
    MVI   : oCW <= 'b__0_0_0_1_0_0_xxxx_0001_10_0_1;
    LSL   : oCW <= 'b__1_0_0_1_0_0_xxxx_1011_xx_0_1;
    LSR   : oCW <= 'b__1_0_0_1_0_0_xxxx_1100_xx_0_1;
    ASR   : oCW <= 'b__1_0_0_1_0_0_xxxx_1101_xx_0_1;
    ITOF  : oCW <= 'b__1_0_1_0_0_0_x010_xxxx_xx_0_1;
    FTOI  : oCW <= 'b__1_0_1_0_0_0_x001_xxxx_xx_0_1;
    FMUL  : oCW <= 'b__1_1_1_0_0_0_x100_xxxx_00_0_1;
    FDIV  : oCW <= 'b__1_1_1_0_0_0_x111_xxxx_00_0_1;
    FADD  : oCW <= 'b__1_1_1_0_0_0_x101_xxxx_00_0_1;
    FSUB  : oCW <= 'b__1_1_1_0_0_0_x110_xxxx_00_0_1;
    FNEG  : oCW <= 'b__0_0_1_0_0_0_xxxx_xxxx_00_0_0;
    FSQRT : oCW <= 'b__1_0_1_0_0_0_x011_xxxx_00_0_1;
    LDL   : oCW <= 'b__0_1_0_0_1_0_xxxx_xxxx_00_0_1;
    LDCI  : oCW <= 'b__0_0_0_1_0_0_xxxx_0001_11_1_1;
    STL   : oCW <= 'b__1_1_0_0_0_1_xxxx_xxxx_00_0_0;
    default: oCW <= 0;
    endcase
end
```

구현 – 스레드

```
module GPPCU_THREAD # (
    parameter
    WORD_BITS   = 10
)
(
    iACLK,
    inRST,

    iCW_DEC,
    iCW_EXEC,
    iCW_WB,

    iINSTR_FCH,
    iINSTR_DEC,
    iINSTR_EXEC,
    iINSTR_WB,

    iLMEMCLK,
    iLMEMSEL,
    iLMEMWREN,
    iLMEMADDR,
    oLMEMRDATA,
    iLMEMWDATA,
    iGMEMDATA,

    oBUSY // For multi-cycle fpu
);
```

위의 블록 다이어그램에서도 잠깐 보인 바와 같이, 스레드의 인터페이스는 위와 같다. 각각의 스테이지에서 동작에 필요한 제어 워드와 인스트럭션을 코어로부터 공급받으며, 외부에서 직접 접근하는 메모리 관련 인터페이스가 있다.

oBUSY 신호는 내부에서 FPU에 의해 멀티사이클 작업이 실행되고 있을 때 세트 된다.

```
GPPCU_THREAD_REGBANK
GPPCU_THREAD_REGBANK_inst(
  .iACLK     (iACLK),
  .iREGASEL
(iINSTR_DEC[INSTR_REGA_5+:5]),
  .iREGBSEL
(iINSTR_DEC[INSTR_REGB_5+:5]),
  .iREGDSEL   (iINSTR_WB
[INSTR_REGD_5+:5]), // FROM PIPE_WRBK
  .oREGA     (dec_reg_a),
  .oREGB     (dec_reg_b),
  .iREGD     (wrbk_reg_d),
  .iWR       (cw_valid_wb &
iCW_WB[CW_REGWR] & wrbk_cond_verified)
);
```

레지스터 뱅크는 디코딩 스테이지에서 지정된 피연산자들을 인출하고, Writeback 스테이지에서 지정된 레지스터를 저장한다. 이 때, Write Enable 신호는 위와 같이 vailidity와 CW, 그리고 condtion verification을 바탕으로 할당된다.

```
assign oREGA
= iWR && iREGASEL == iREGDSEL ? iREGD :
rg[iREGASEL];
assign oREGB
= iWR && iREGBSEL == iREGDSEL ? iREGD :
rg[iREGBSEL];
```

레지스터 뱅크는 내부적으로 위와 같이, WriteBack 스테이지에서 입력되는 Valid한 데이터에 대해 인출 요청이 들어오는 경우, 데이터의 Preview를 허용하고 있다. 이를 통해 Data가 Invalid한 스테이지를 하나 줄일 수 있다.

이렇게 인출된 REGISTER A와 REGISTER B는 각각 Execution 스테이지의 피연산자로 사용된다. 단, Register A는 그대로 사용되지만, B는 제어 워드의 신호에 따라 멀티플랙싱 되어 전달된다.

```
2'd0: dec_opr_b_mux <=
dec_reg_b+iINSTR_DEC[INSTR_IMM0_7+:7];
2'd1: dec_opr_b_mux <=
iINSTR_DEC[INSTR_IMM1_12+:12];
2'd2: dec_opr_b_mux <=
iINSTR_DEC[INSTR_IMM2_17+:17];
2'd3: dec_opr_b_mux <= iGMEMDATA;
```

위와 같이 Instruction Model에 따라 B의 형태가 바뀌며, 특히 2'd3의 경우 위에서도 언급한 GMEMDATA가 인가되는 것을 볼 수 있다.

```
reg [31:0]
    exec_oprand_a, exec_oprand_b;
always @(posedge iACLK) exec_oprand_a
    <= dec_reg_a;
always @(posedge iACLK) exec_oprand_b
    <= dec_opr_b_mux;
```

레지스터 A와 멀티플랙싱 된 B는 이렇게 다음 스테이지의 플랫폼으로 Unconditional하게 로드 된다. 로드되는 값들은 모두 통제되는 컨트롤 워드에 의해 결정되므로 이 동작은 안전하다.

Execution 단계는 매우 중요한 동작을 다수 포함하고 있다.

메모리의 읽기 및 쓰기 동작도 Execution 스테이지에서 일어난다. exec_oprand_a를 데이터로, exec_oprand_b를 어드레스로 사용하며, 쓰기 동작은 Execution 스테이지에서 마무리가 되며, 읽기 동작을 통해 출력된 데이터는 Writeback 스테이지로 전달된다.

```
DPRAM_PARAM #(
    .DBW(32),
    .DEPTH(1 << WORD_BITS)
) DPRAM_PARAM_local_memory
(
    // For external interface
    .iPA_CLK    (iLMEMCLK),
    .iPA_ADDR   (iLMEMADDR),
    .iPA_WR     (iLMEMWREN & iLMEMSEL),
    .iPA_WDATA  (iLMEMWDATA),
    .oPA_RDATA  (oLMEMRDATA),

    // For internal use
    .iPB_CLK    (iACLK),
    .iPB_ADDR   (exec_oprand_b),
    .iPB_WR             (cw_valid_exec   &
iCW_EXEC[CW_LMEM_WR]                    &
exec_cond_verified),
    .iPB_WDATA  (exec_oprand_a),
    .oPB_RDATA  (wrbk_locmem_dat) // Data
read performs unconditionally.
);
```

```
reg  [ 4:0] exec_sreg = 0;
reg  [31:0] wrbk_alu_q = 0, wrbk_fpu_q =
0;
wire [31:0] alu_q, fpu_q;
wire [ 4:0] alu_sreg, fpu_sreg/*No
valid status register.*/;
assign fpu_sreg = 0;
wire [ 4:0] pending_sreg =
iCW_EXEC[CW_FPOP] ? fpu_sreg :
alu_sreg;
```

이들은 Execution 스테이지에서 결정되는 신호이다. 5비트(VCNZ, 1 bit reserved)의 상태 레지스터가 여기에서 관리되며, 스루풋을 조금이라도 줄이기 위해 FPU와 ALU에서 계산된 결과의 멀티플렉싱은 Writeback 스테이지로 위임된다.

```
exec_sreg  <= ~inRST ? 0 : cw_valid_exec
& exec_cond_verified &
iINSTR_EXEC[INSTR_S] ? pending_sreg :
exec_sreg;
```

상태 레지스터는 해당 작업이 valid하고(COND 플래그와 상태 레지스터의 비교를 통해) 인스트럭션의 S 플래그가 지정되었을 경우에만 업데이트된다.

```
GPPCU_ALU #(
    .BW     (32)
) GPPCU_ALU_inst
(
    .iA     (exec_oprand_a),
    .iB     (exec_oprand_b),
    .iC     (exec_sreg[SREG_C]),
    .iOP    (iCW_EXEC[CW_ALOPC0+:4]),

    .oV     (alu_sreg[SREG_V]),
    .oC     (alu_sreg[SREG_C]),
    .oN     (alu_sreg[SREG_N]),
    .oZ     (alu_sreg[SREG_Z]),
    .oQ     (alu_q)
);
```

ALU의 인스턴스이다. 특별할 것은 없는 구현이다.

```
wire        fp_busy;
reg         fp_start;
wire        fp_done;
reg[1:0]    fp_stage = 0;
reg[31:0]   fpu_da, fpu_db;
localparam [1:0]
    FP_IDLE     = 0,
    FP_RUN      = 1,
    FP_BUSY     = 2,
    FP_DONE     = 3
;
```

위에서 언급한 FPU 상태 머신 레지스터이다. FPU가 여러 사이클에 걸쳐 실행되는 동안, 파이프라인을 안전하게 블록하기 위해 사용한다.

```
assign fp_busy
= cw_valid_exec && fp_stage != FP_DONE
&& iCW_EXEC[CW_FPOP];
```

Execution 스테이지에 FPU 작업이 로드되는 즉시 활성화되며, 이는 아래의 상태 머신에 따라 동작한다. (다음 페이지)

```
case(fp_stage)
    FP_IDLE: begin
        fp_stage    <= cw_valid_exec &
iCW_EXEC[CW_FPOP] ? FP_RUN : FP_IDLE;
        fpu_da      <= exec_oprand_a;
        fpu_db      <= exec_oprand_b;
    end
    FP_RUN: begin
        fp_stage    <= FP_BUSY;
        fp_start    <= 1'b1;
    end
    FP_BUSY: begin
        fp_stage    <= fp_done ?
FP_DONE : FP_BUSY;
        wrbk_fpu_q  <= fp_done ? fpu_q   :
wrbk_fpu_q ;
        fp_start    <= 1'b0;
    end
    FP_DONE: begin
        fp_stage    <= FP_IDLE;
        fp_start    <= 1'b0;
    end
endcase
```

대기 상태에서 시작 클럭을 생성하고, 동작이 완료될 때까지 대기한 뒤 출력 데이터를 로드 하는 식이다. 이 과정에서 약 세 클럭의 오버헤드가 발생하게 된다.

아래는 FPU의 인스턴스이다.

```
GPPCU_MC_FPU GPPCU_MC_FPU_inst(
    .clk        (iACLK),
    .clk_en     (1'b1),
    .dataa      (fpu_da),
    .datab      (fpu_db),
    .n          (iCW_EXEC[CW_FPOPC0+:3]),
    .reset      (0),
    .reset_req  (0),
    .start      (fp_start),
    .done       (fp_done),
    .result     (fpu_q)
); /*/
```

위와 같은 신호로 인가되어 있다.

Writeback 스테이지의 동작은 위의 REGBANK에 대한 설명과 같으므로 이곳에서는 생략한다.

인터페이스

위에서 설명한 바에 따라, 병렬 연산기의 동작을 요약하면

1. 데이터를 스레드 로컬 메모리에 업로드
2. 명령어를 공급
3. 작업이 완료된 데이터를 로컬 메모리로부터 다운로드

이라고 할 수 있다.

주로 사용되는 환경을 가정한다면 별도의 CPU가 존재하고, 하드웨어 가속기로 병렬 연산기를 사용하는 시나리오가 가장 보편적이다.

그러나 그 상황에서 병렬연산기 코어만을 버스에 연결해 두면, CPU는 계속해서 병렬 연산기가 인스트럭션을 받을 수 있는지 동기적으로 확인하고, 새로운 인스트럭션을 직접 공급해야 한다.

이는 큰 비효율을 초래하므로, Instruction Memory의 역할을 수행하는 공급기를 별도로 설계하였다.

```
module GPPCU_TEST_QUEUE #(
    parameter
    QBW                 = 12,
    GBW                 = 10,
    NUM_THREAD           = 24
)
(
    iACLK               ,
    inRST               ,
    iCMD                ,
    iDATA               ,
    oDATA               ,
    oDONE
);
```

처음엔 테스트로 시작한 모듈인데, 이름을 아직 바꾸지 않았다. 실제로는 INSTRUCTION SUPPLIER가 적절한 이름으로 보인다.

이 모듈의 인터페이스는 단순한데, 특히 NIOS의 32비 트 PIO 입출력만으로도 동작할 수 있게끔 설계하였다.

iDATA, oDATA, iCMD 세 신호 모두 32비트 PIO 입출력 신호이며, 다음과 같이 해석한다.

```
ABOUT CMD
    1       7       8       16 bits
    CLK     WPARAM  LPARAM  COMMAND

    WPARAM  NAME                LPARAM  COMMAND         iDATA       oDATA
    0       PUSH INSTRUCTION    X       X               INSTR
    1       READ LOCAL MEMORY   THREAD  ADDRESS         X           OUTPUT
    2       WRITE LOCAL MEMORY  THREAD  ADDRESS         INPUT       X
    3       SET GLOBAL MEMORY   X       ADDRESS         INPUT
    4       OPERATION COMMAND                           X           STATUS
        LPARAM  COMMAND             NAME
        0       1 = RUN, 0 = STOP   RUN OR STOP
        1       NUM CYCLES          SET CYCLES
        2       OFST PER CYCLE      SET OFST PER CYCLES
        3       X                   RESET PROGRAM
```

이 인터페이스는 단순히 프로그램을 저장하는 것만이 아니라, 자동으로 프로그램을 '태스크' 단위로 반복적으로 공급하게 된다.

태스크는 최소한의 작업 단위이며, 예를 들어 백 개의 Vertex에 대해 반복 작업을 한다고 가정하면, 하나의 Vertex에 하나의 태스크를 할당하게 된다.

태스크는 Thread에 분산 배치되며, 즉 본 프로젝트에서 스레드는 24개가 인스턴스화 되므로 0번부터 23번까지 스레드 0…23번에 순서대로 태스크가 할당된 뒤, 24번부터 다시 스레드 0번으로 돌아가 태스크를 할당하는 방식이다.

즉, 태스크는 아래와 같은 방식으로 할당되며, 실행은 좌에서 우로, 즉 동시에 24개의 태스크가 수행된다. 이 단위를 Cycle이라 부르고, 100개의 태스크를 수행해야 한다면 즉 전체적으로 (100 + 23) / 24 = 5 번의 태스크 사이클이 수행되어야 한다.

```
-------> Execution
THREAD 0 [TASK0] [TASK24] [TASK48] ..
THREAD 1 [TASK1] [TASK25] [TASK49] ..
THREAD 2 [TASK2] [TASK26] [TASK50] ..
THREAD 3 [TASK3] [TASK27] [TASK51] ..
THREAD 4 [TASK4] [TASK28] [TASK52] ..
```

각각의 태스크에는 스레드의 로컬 메모리 공간이 할당되므로, 각 태스크를 작업할 때는 시작 주소를 조금씩 오프셋 시키게 된다.

위와 같이 태스크는 서로 다른 메모리 오프셋에 대해 같은 프로그램을 반복적으로 수행하게 되므로, 하나의 태스크를 수행하는 프로그램만 넣어둔 뒤 한 번 사이클을 돌 때마다 메모리 오프셋만 변경시켜 반복시키면 메모리 공간을 최대한으로 절약하고, 효율적인 비동기 작업을 이끌어낼 수 있다.

```
reg     [31:0]      cur_task_ofst;
reg     [31:0]      sz_per_task;
reg     [11:0]      num_cycles;
reg     [11:0]      cur_cycle_idx;
```

이러한 프로그램의 상태는 위와 같은 내부 변수들로 관리되며, 특히 cur_task_ofst 변수는 메모리의 오프셋을 결정하는 중요한 역할을 수행한다.

이 모듈은 주로 두 개의 클록 펄스에 의해 제어되는데, 하나는 CMD 입력의 최상위 비트 클럭이고, 다른 하나는 통상적인 ACLK 입력이다.

```
always @(posedge opclk or negedge
inRST) begin
    if(~inRST) begin
        pmem_end <= 0;
        pmem_running <= 0;
    end
    else if(wparam == OPR_INSTR) begin
        if(~pmem_running) begin
            pmem_end <= pmem_end + 1;
        end
    end
    else if(wparam == OPR_COMMAND) begin
        case (lparam)
            LPM_RUNSTOP: begin
                pmem_running <=
command[0];
            end
            LPM_RESETPRG: begin
                pmem_end <= 0;
                pmem_running <= 0;
            end
            LPM_NUMCYCLE: begin
                num_cycles <= command;
            end
            LPM_SZPERCYCLE: begin
                sz_per_task <= command;
            end
        endcase
    end
end
```

CMD 입력의 클럭을 opclk라 부른다.

pmem_end는 언제 사이클이 멈춰야 할지 판단하는 척도로 사용되며, 현재 입력된 인스트럭션의 개수를 나타낸다.

pmem_running은 이름을 다소 모호하게 지었는데, 실제론 프로그램의 실행 여부를 결정하는 플래그이다. 프로그램 실행 도중에는 프로그램을 수정할 수 없게끔 구성되어 있다.

이외에도 외부에서 직접 몇 번의 사이클을 수행하고, 각 태스크가 차지하는 메모리 공간이 몇 워드인지를 지정해줄 수 있다. 이들 파라미터는 반복 작업을 수행하는 데 대단히 중요한 역할을 한다.

```
always @(posedge iACLK) begin
    if(pmem_running & inRST) begin
        if(cur_cycle_idx == num_cycles)
begin
            program_done <= pmem_running;
            // to generate instr_invalid
signal.
        end
        else if(pmem_head < pmem_end)
begin
            if(instr_ready) begin
                pmem_head <= pmem_head +
1;
            end
        end
        else begin
            pmem_head <= 0;
            cur_task_ofst <=
cur_task_ofst + sz_per_task;
            cur_cycle_idx <=
cur_cycle_idx + 1;
        end
    end
    else begin
        program_done    <= 0;
        pmem_head       <= 0;
        cur_task_ofst   <= 0;
        cur_cycle_idx   <= 0;
    end
end
```

다음은 ACLK로 구동되는 로직이다. 특히 주목할 부분은 현재 프로그램의 주소를 나타내는 pmem_head가 pmem_end에 도달했을 때의 동작으로, pmem_head를 0으로 돌린 뒤 task의 시작 주소를 오프셋하고, 현재 사이클 인덱스를 하나 올린다.

이후 다음 클럭에서 현재 사이클의 인덱스가 목표 사이클에 도달했는지를 검사하고, 도달했다면 프로그램 종료를 알린다. 중요한 점은, 프로그램은 초기화되지 않으므로 몇 번이고 재사용이 가능하다는 것이다.

이후 pmem_running을 비활성화시키면 다시 프로그램의 진행 상태가 초기화된다. pmem_running이 내려간 상태에서 ACLK 입력이 들어와야 하므로, opclk가 지나치게 빠르면 위의 로직이 pmem_running이 reset되었음을 놓칠 수 있으므로, 몇 pmem_running을 리셋한 이후 클럭 정도 의도적인 딜레이를 주는 것이 좋다.

Figure 2. 파이프라인 시뮬레이션

상당히 중요한 부분인데, 태스크라는 개념은 추상화 단계를 거친 내용이다. 위에서도 보았지만 GPPCU CORE 자체는 태스크 등의 기능에 대한 지원이 일절 없다.

이를 구현하기 위해서 이용하는 것이 바로 GMEM이다.

GMEM은 외부에 어드레스를 요청하고 클럭 펄스를 내보내는 방식으로 동작하므로, 외부 회로에서 얼마든지 GMEM의 어드레스를 디코딩해 임의의 GMEMDATA를 전달할 수 있다.

따라서, 스레드로 하여금 현재의 태스크 메모리 오프셋 위치를 알 수 있도록 아래와 같은 로직을 정의하였다.

```
assign iGMEM_WDATA = oGMEM_ADDR == 0 ?
cur_task_ofst : gmem_dat;
```

즉, 내부에서 어드레스 0에 대한 GMEM을 요청하면 태스크의 오프셋을 반환하는 것이다.

```
GPPCU_CORE # (
    .NUM_THREAD (NUM_THREAD),
    .WORD_BW    (9)
) GPPCU_CORE_inst
(
    .iACLK              (iACLK),
// Instruction should be valid on
ACLK's rising edge.
    .inRST              (inRST),
    .oIDLING            (gppcu_idle),
// Doing nothing.
    .iINSTR             (program),
    .iINSTR_VALID       (instr_valid),
    .oINSTR_READY       (instr_ready),
// Replaces instruction address.
    .iLMEM_CLK          (opclk),
    .iLMEM_THREAD_SEL   (lparam), //
Thread selection input
    .iLMEM_ADDR         (command),
    .iLMEM_WDATA        (iDATA),
    .oLMEM_RDATA        (data_out),
    .iLMEM_RD           (wparam ==
OPR_RDL),
    .iLMEM_WR           (wparam ==
OPR_WRL),
    .oGMEM_ADDR         (oGMEM_ADDR),
    .iGMEM_WDATA        (iGMEM_WDATA),
    .oGMEM_CLK          (oGMEM_CLK)
);
```

코어는 모듈 내부에 이렇게 인스턴스화되어 있다.

여기까지가 베릴로그로 구현된 하드웨어이다. 이제 이 장치를 테스트하고 구동하는 드라이버를 작성하기 위해, NIOS II Synthesizable Processor를 프로젝트에 병합하였다. 시스템은 다음과 같다.

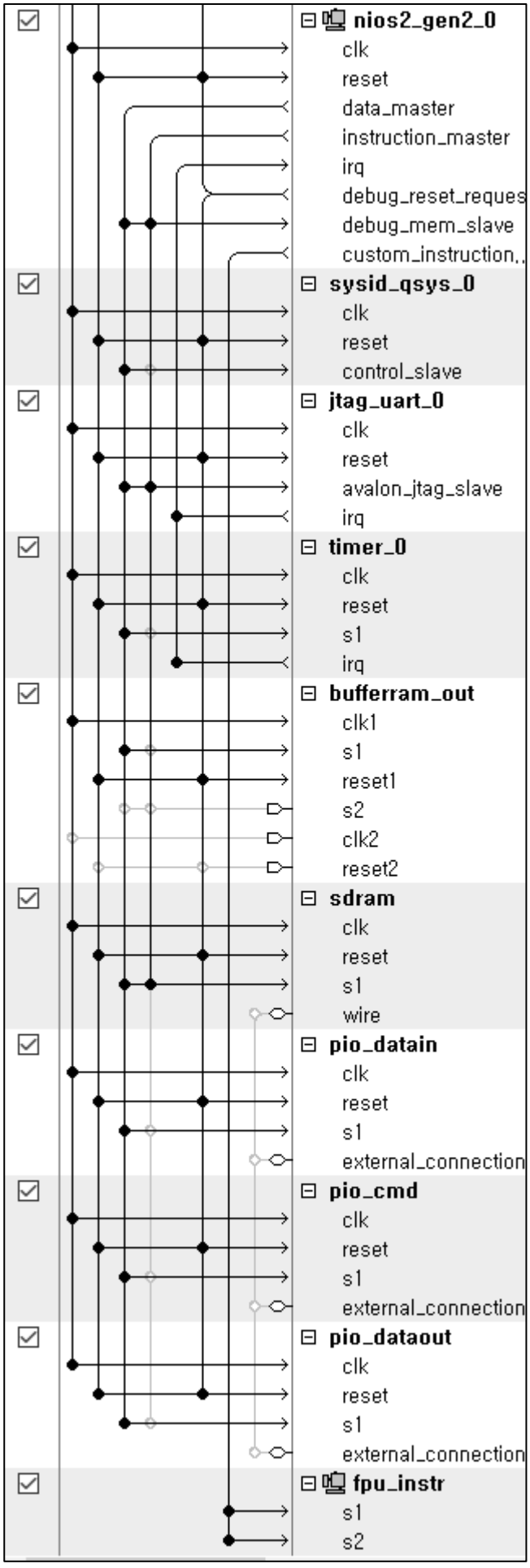
가장 먼저 nios cpu가 있다. 150MHz로 PLL을 이용해 분주된 클럭을 인가하였고, 월드 행렬을 만드는 것은 CPU에서 하게 되므로, 빠른 FP 연산을 위한 커스텀 인스트럭션 모듈을 추가하였다.

다음으로 VEEK-MK2 보드에 실장된 128MB SDRAM을 사용하기 위한 인터페이스를 구성하였고, 2-port 메모리를 사용해 LCD 버퍼를 CPU에서 접근할 수 있게끔 만들었다.

PIO 신호는 데이터 입/출력과 커맨드 출력의 세 종류가 있으며, 모두 32비트 너비를 갖는다.

```
resource/TOP.v
resource/QSYS/SYSTEM.qsys
resource/HDL/GPPCU/GPPCU_TEST_QUEUE.v
resource/GENERATED/GPPCU_MC_FPU/synthesis/GPPCU_MC_FPU.qip
resource/HDL/GPPCU/GPPCU_COND_VERIFIER.v
resource/HDL/GPPCU/GPPCU.v
resource/HDL/GPPCU/GPPCU_THREAD_REGBANK.v
resource/HDL/DEVICE_INDEPENDENT/DPRAM_PARAM.v
resource/HDL/GPPCU/PRIOIRTY_ENCODER_PARAM.v
resource/HDL/GPPCU/GPPCU_THREAD.v
resource/HDL/GPPCU/GPPCU_STALL_GEN.v
resource/HDL/GPPCU/GPPCU_INSTR_QUEUE.v
resource/HDL/GPPCU/GPPCU_INSTR_DEC.v
resource/HDL/GPPCU/GPPCU_FPU_MULTICYCLE.v
resource/HDL/GPPCU/GPPCU_CORE.v
resource/HDL/GPPCU/GPPCU_ALU.v
resource/HDL/GPPCU/ENCODER_PARAM.v
resource/HDL/GPPCU/DECODER_PARAM.v
resource/HDL/LCD/LCD_CON.v
resource/HDL/LCD/LCD.v
resource/GENERATED/BUFFER_TFT.qip
resource/GENERATED/SDRAM_PLL.qip
resource/DOC/note.txt
```

Figure 3. 작성된 소스 코드



```
 SYSTEM SYSTEM_inst(
    .bufferram_out_address
(buff_addr),
    .bufferram_out_chipselect
(bufferram_out_chipselect),
    .bufferram_out_clken        (1),
    .bufferram_out_write        (/*MUST
SET 0 LATER*/0),
    .bufferram_out_readdata
(bufferram_out_readdata),
    .bufferram_out_writedata    (/*MUST
SET 0 LATER*/0),//buff_addr),
    .bufferram_out_byteenable   (2'b11),
    .bufferram_rst_reset
(bufferram_rst_reset),
    .buffram_clk_clk
(buffram_clk_clk),
    .clk_clk
(iCLK50MHz),
    .pio_cmd_out_export
(gppcu_cmd),      //
pio_cmd_out.export
    .pio_data_in_export
(gppcu_dataout),      //
pio_data_in.export
    .pio_data_out_export
(gppcu_datain),     //
pio_data_out.export
    .sdram_wire_addr
(DRAM_ADDR),
    .sdram_wire_ba              (DRAM_BA),
    .sdram_wire_cas_n
(DRAM_CAS_N),
    .sdram_wire_cke
(DRAM_CKE),
    .sdram_wire_cs_n
(DRAM_CS_N),
    .sdram_wire_dq              (DRAM_DQ),
    .sdram_wire_dqm
(DRAM_DQM),
    .sdram_wire_ras_n
(DRAM_RAS_N),
    .sdram_wire_we_n
(DRAM_WE_N)
);
```

탑 모듈의 시스템 인스턴스.

GPPCU 모듈 인스턴스. 상당히 간단한 인터페이스를 갖는다.

```
GPPCU_TEST_QUEUE #(
    .QBW                (11),
    .GBW                (6),
    .NUM_THREAD          (24)
) GPPCU_TEST_QUEUE_inst(
    .iACLK              (iCLK50MHz),
    .inRST              (inRST),
    .iCMD               (gppcu_cmd),
    .iDATA              (gppcu_datain),
    .oDATA              (gppcu_dataout)
);
```

```
LCD LCD_inst
(
    .clk                (iCLK50MHz),
    .rst_               (inRST),
    .bBL                (1),
// Backlight en
    .bDTH               (0),
// Dithering en
    .oBRAM_CLK           (bram_clk),
// to VRAM
    .iCOLOR
(buffer_to_tft_color),            //
from VRAM
    .oHADDR             (tft_haddr),
    .oVADDR             (tft_vaddr),
    .oADDR
(bufferram_out_address),
    .oLCDRGB            (LCDRGB),
// to LCD device output
    .oLCDCON            (LCDCON[13:3]),
    .oDE               (tft_de),
    .oHSYNC             (tft_hsync),
    .oVSYNC             (tft_vsync)
);
```

LCD 컨트롤러 인스턴스. 내부에 종/횡 및 전체 어드레스 카운터를 포함하고 있다.

NIOS에서 C언어를 통해, 범용 연산기의 드라이버를 작성하였다.

```
typedef uint32_t swk_gppcu_data_t;
typedef uint32_t swk_gppcu_instr_t;
struct swk_gppcu
{
    swk_gppcu_instr_t*   marr;
    int32_t     mcap;
    int32_t     mnum;
    int32_t     mnumthr;
    int32_t     mtaskcycle;
    int32_t     mtaskmem;
    int32_t     ro_taskmaxcycle;
    int32_t     ro_numtask;
    int32_t     ro_max_word_per_thread;

    uint32_t    MMAP_DATOUT;
    uint32_t    MMAP_DATIN;
    uint32_t    MMAP_CMDOUT;
};
```

marr 힙 메모리에 명령어를 담는다.

mcap marr의 최대 크기

mnum 현재 명령어 카운트

mnumthr GPPCU의 스레드 카운트. 상수

mtaskcycle 태스크 개수를 기반으로 계산된 사이클 개수

Mtaskmem 태스크당 할당된 메모리

ro_taskmaxcycle 태스크 메모리 기반으로, 최대 사이클 개수.

ro_numtask 태스크 개수

ro_max_word… 스레드당 최대 워드 개수. 현재 모델에선 512개

아래 세 uint32값은 PIO 베이스 어드레스이다.

```
void gppcu_init(
    swk_gppcu_t* const pp,
    int32_t num_threads,
    int32_t Capacity,
    int32_t MaxWordPerThread,
    uint32_t CMDOUT,
    uint32_t DATOUT,
    uint32_t DATIN
);

void gppcu_destroy(swk_gppcu_t* const pp);
void
gppcu_program_autofeed_device_parallel( swk_g
ppcu_t const* const pp );
void gppcu_run_autofeed_device( swk_gppcu_t
const* const pp );

void gppcu_clear_instr(swk_gppcu_t* const
pp);

void gppcu_init_task( swk_gppcu_t* const pp,
uint8_t WordsPerTask, uint16_t NumTasks );
void      gppcu_init_task_parallel(swk_gppcu_t
*const  pp,  uint8_t  WordsPerTask,  uint16_t
NumTasks);
bool gppcu_is_done( swk_gppcu_t* const pp,
uint16_t* cycles_left );
```

생성과 소멸, 명령어 프로그램 등을 위한 인터페이스를 제공한다.

```
void gppcu_write(
    swk_gppcu_t* pp,
    swk_gppcu_data_t const* const data,
    uint8_t ElementSizeInWords,
    uint32_t ofst // Means local space offset
on task domain. Units in word
);
void gppcu_read(
    swk_gppcu_t* pp,
    swk_gppcu_data_t * const dst,
    uint32_t Capacity,
    uint8_t ElementSizeInWords,
    uint32_t ofst
);
```

가장 중요한 인터페이스로, 데이터를 태스크 단위로 분배해 로컬 메모리에 프로그램한다.

아래와 같은 로직으로, 먼저 수평으로 스레드에 메모리를 채우고, 한 사이클을 돌면 베이스를 오프셋한 뒤 이를 반복한다. (다음 페이지)

```
void gppcu_write(
    swk_gppcu_t* pp,
    swk_gppcu_data_t const* const data,
    uint8_t ElementSizeInWords,
    uint32_t ofst // Means local space offset
on task domain. Units in word
)
{
    int idx_cycle = 0;
    int cycle_based_offset = 0;
    int idx_task, idx_thread;
    swk_gppcu_data_t const* head = data;

    passert( ofst + ElementSizeInWords < pp-
>mtaskmem, "Task memory overflow" );
    // Write data per thread
    for ( idx_task = 0, idx_thread = 0;
idx_task < pp->ro_numtask; ++idx_task )
    {
        // @todo.
        // write task_data
        int elem_idx;
        for ( elem_idx = 0; elem_idx <
ElementSizeInWords; ++elem_idx )
        {
            gppcu_data_wr( pp->MMAP_CMDOUT,
pp->MMAP_DATOUT, idx_thread,
cycle_based_offset + ofst + elem_idx,
*head++ );
        }

        ++idx_thread;
        if ( idx_thread == 24 )
        {
            idx_thread = 0;
            ++idx_cycle;

            cycle_based_offset += pp-
>mtaskmem;
        }
    }
}
```

Read 함수도 비슷한 방법을 사용한다.

Gppcu 드라이버는 gppcu 프로그램을 구성하기 위한 함수를 제공한다. 아래의 함수들을 호출할 때마다, 해당하는 명령이 하나씩 gppcu::marr에 누적되고, 이후 'program'에 해당하는 함수를 호출했을 때 일괄적으로 gppcu에 로드된다.

```
static inline void gppcu_nop(
    swk_gppcu_t* const pp )
{
    gppcu_put_instr( pp, 0 );
}

static inline void gppcu_arith_s(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    bool s,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER regb,
    int8_t imm7 )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
opr, s != 0, regd, 0, imm7, regb )
    );
}
static inline void gppcu_arith_0(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    bool s,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER rega )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_B( cond,
opr, s != 0, regd, rega, 0 )
    );
}

static inline void gppcu_mvi(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    bool s,
    GPPCU_REGISTER regd,
    int32_t imm17 )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_C( cond,
OPR_C_MVI, s != 0, regd, imm17 )
    );
}

static inline void gppcu_arith_a(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    bool s,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER rega,
    GPPCU_REGISTER regb,
    int8_t imm7 )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
opr, s != 0, regd, rega, imm7, regb )
    );
}

static inline void gppcu_arith_b(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    bool s,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER rega,
    int16_t imm12 )
{
```

```
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_B( cond,
opr, s != 0, regd, rega, imm12 )
    );
}
static inline void gppcu_fp_arith(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER rega,
    GPPCU_REGISTER regb )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
opr, 0, regd, rega, 0, regb )
    );
}
static inline void gppcu_fp_0(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_OPERATION opr,
    GPPCU_REGISTER regd,
    GPPCU_REGISTER rega )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
opr, 0, regd, rega, 0, 0 )
    );
}
static inline void gppcu_ldl(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_REGISTER dest,
    GPPCU_REGISTER addr,
    uint8_t ofst )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
OPR_LDL, 0, dest, 0, ofst, addr )
    );
}
static inline void gppcu_stl(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_REGISTER data,
    GPPCU_REGISTER addr,
    uint8_t ofst )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_A( cond,
OPR_STL, 0, 0, data, ofst, addr )
    );
}
static inline void gppcu_ldci(
    swk_gppcu_t* const pp,
    GPPCU_CONDTION cond,
    GPPCU_REGISTER dest,
    uint32_t addr )
{
    gppcu_put_instr(
        pp,
        GPPCU_ASSEMBLE_INSTRUCTION_C( cond,
OPR_LDCI, 0, dest, addr )
    );
}
```

GPPCU_CONDITON과 GPPCU_OPERATION은 다음과 같이 정의되어 있다. 모두 전역 범위 enum 상수이다.

```
typedef enum GPPCU_OPERATION
{
        OPR_NOP     =  0,
        OPR_S_MOV     =  1,
        OPR_S_MVN     =  2,
        OPR_A_ADC     =  3,
        OPR_A_SBC     =  4,
        OPR_A_AND     =  5,
        OPR_A_ORR     =  6,
        OPR_A_XOR     =  7,
        OPR_B_ADI     =  8,
        OPR_B_SBI     =  9,
        OPR_C_MVI     = 10,
        OPR_0_LSL     = 11,
        OPR_0_LSR     = 12,
        OPR_0_ASR     = 13,
        OPR_0_ITOF    = 14,
        OPR_0_FTOI    = 15,
        OPR_A_FMUL    = 16,
        OPR_A_FDIV    = 17,
        OPR_A_FADD    = 18,
        OPR_A_FSUB    = 19,
        OPR_X_FNEG    = 20,
        OPR_0_FSQRT   = 21,
        OPR_LDL     = 22,
        OPR_LDCI    = 23,
        OPR_STL     = 24
} GPPCU_OPERATION;

typedef enum GPPCU_CONDTION
{
        COND_ALWAYS = 0,
        COND_NEVER  = 1,
        COND_C      = 2,
        COND_NC     = 3,
        COND_Z      = 4,
        COND_NZ     = 5,
        COND_V      = 6,
        COND_NV     = 7,
        COND_N      = 8,
        COND_NN     = 9,
        COND_NEG    = 10,
        COND_POS    = 11,
        COND_EQ     = 12,
        COND_GR     = 13,
        COND_LT     = 14,
} GPPCU_CONDTION;
```

이 함수를 그대로 사용하면 코드가 상당히 verbose 해지기 때문에, 매크로 형태로 간단한 헬퍼를 제공한다.

```
#define gp_set_gppcu_ptr(ptr) swk_gppcu_t*
const GPPCU_INSTANCE____ = ptr
#define gp_fmul(rd, ra, rb)
gppcu_fp_arith(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_FMUL, rd, ra, rb)
#define gp_fadd(rd, ra, rb)
gppcu_fp_arith(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_FADD, rd, ra, rb)
#define gp_fsub(rd, ra, rb)
gppcu_fp_arith(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_FSUB, rd, ra, rb)
#define gp_fdiv(rd, ra, rb)
gppcu_fp_arith(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_FDIV, rd, ra, rb)
#define gp_fsqrt(rd, ra)
gppcu_fp_0(GPPCU_INSTANCE____, COND_ALWAYS,
OPR_0_FSQRT, rd, ra)
#define gp_ftoi(rd, ra)
gppcu_fp_0(GPPCU_INSTANCE____, COND_ALWAYS,
OPR_0_FTOI, rd, ra)
#define gp_itof(rd, ra)
gppcu_fp_0(GPPCU_INSTANCE____, COND_ALWAYS,
OPR_0_ITOF, rd, ra)

#define gp_ldl(dest, addr, ofst)
gppcu_ldl(GPPCU_INSTANCE____, COND_ALWAYS,
dest,addr,ofst)
#define gp_stl(data, addr, ofst)
gppcu_stl(GPPCU_INSTANCE____, COND_ALWAYS,
data,addr,ofst)
#define gp_ldci(rd, addr)
gppcu_ldci(GPPCU_INSTANCE____, COND_ALWAYS,
rd, addr)

#define gp_nop()
gppcu_nop(GPPCU_INSTANCE____)
#define gp_mov(to, from, add)
gppcu_arith_s(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_S_MOV, false, to, from, add)
#define gp_mvi(to, imm17)
gppcu_mvi(GPPCU_INSTANCE____,     COND_ALWAYS,
false, to, imm17)
#define gp_adc(rd, ra, rb, imm7)
gppcu_arith_a(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_ADC, false, rd, ra, rb,
imm7)
#define gp_sbc(rd, ra, rb, imm7)
gppcu_arith_a(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_SBC, false, rd, ra, rb,
imm7)
#define gp_and(rd, ra, rb, imm7)
gppcu_arith_a(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_AND, false, rd, ra, rb,
imm7)
#define gp_orr(rd, ra, rb, imm7)
gppcu_arith_a(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_ORR, false, rd, ra, rb,
imm7)
#define gp_xor(rd, ra, rb, imm7)
gppcu_arith_a(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_A_XOR, false, rd, ra, rb,
imm7)
#define gp_adi(rd, ra, imm12)
gppcu_arith_b(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_B_ADI, false, rd, ra,
imm12 )
#define gp_sbi(rd, ra, imm12)
gppcu_arith_b(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_B_SBI, false, rd, ra,
imm12 )
#define gp_lsl(rd, ra)
gppcu_arith_0(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_0_LSL, false, rd, ra)
#define gp_lsr(rd, ra)
gppcu_arith_0(GPPCU_INSTANCE____,
COND_ALWAYS, OPR_0_LSR, false, rd, ra)
#define             gp_asr(rd,             ra)
gppcu_arith_0(GPPCU_INSTANCE____, COND_ALWAYS,
OPR_0_ASR, false, rd, ra)
```

이를 이용해 작성한 프로그램은 아래와 같다.

```
for ( col = 0; col < 4; ++col ) {
    gp_mvi( acc, 0 );
    for ( row = 0; row < 4; row++ ) {
        gp_ldci( loader,
COFST_MAT4_WORLD_VIEW_PROJ + col + row * 4 );
        gp_fmul( calc, rx + row, loader );
        gp_adi( loader, acc, 0 );
        gp_fadd( acc, loader, calc );
    }
    gp_stl( acc, REGPIVOT, OFST_TABLE + col );
}
```

위의 코드는 gppcu 구조체에 벡터-행렬 곱을 수행하는 프로그램을 기록한다.

Gppcu는 분기 연산을 지원하지 않으므로, 모든 프로그램은 외부에서 인라이닝하고 Loop unrolling을 수행해주어야 한다.

즉, gppcu의 프로그래밍 과정은 아래와 같다.

```
swk_gppcu_t gppcu;

gppcu_init( &gppcu, 24, 1024, 512,
PIO_CMD_BASE, PIO_DATAOUT_BASE,
PIO_DATAIN_BASE );
gppcu_init_task( &gppcu, 5, 240 );

gppcu_clear_instr( &gppcu );
```

Gppcu 인스턴스를 초기화하고, 태스크를 지정한다. 만약 병렬 연산을 할 요소의 개수가 240개라면, 태스크 또한 240개를 지정하면 된다.

각 태스크에 할당할 메모리도 동시에 지정한다.

이후 인스트럭션을 초기화하고, 예의 인스트럭션 함수를 호출할 때마다 명령어가 하나씩 기록된다.

```
gp_set_gppcu_ptr( &gppcu );

gp_mvi( 0x5, 3 );
gp_mvi( 0x6, 4 );

gp_itof( 0x0, 0x5 );
gp_itof( 0x1, 0x6 );

gp_fmul( 0x2, 0x1, 0x0 );
gp_fdiv( 0x3, 0x1, 0x0 );
gp_fsqrt( 0x4, 0x2 );

gp_stl( 0x0, REGPIVOT, 0 );
gp_stl( 0x1, REGPIVOT, 1 );
gp_stl( 0x2, REGPIVOT, 2 );
gp_stl( 0x3, REGPIVOT, 3 );
gp_stl( 0x4, REGPIVOT, 4 );
```

위와 같이 헬퍼 매크로를 응용해 어셈블리 프로그램을 작성할 수 있다. REGPIVOT은 가장 높은 레지스터로, 현재 레지스터의 오프셋을 시뮬레이션한다. 스레드가 로컬 메모리와 정상적으로 통신하기 위해서는 REGPIVOT 내부의 값을 수정하지 말아야 한다.

메모리에 대한 저장 연산은 전부 REGPIVOT에 대한 오프셋을 바탕으로 수행하게 된다.

```
gppcu_program_autofeed_device_parallel( &gppc
u );
gppcu_run_autofeed_device( &gppcu );
```

이후 program 함수를 호출해주면 프로그램이 gppcu에 로드되고, run_... 함수를 호출하면 동작을 시작한다. 이 함수는 동작을 block하지 않으므로, gppcu가 프로그램을 수행하는 동안 다른 작업을 할 수 있다.

```
gppcu_destroy( &gppcu );
```

사용을 마친 gppcu는 메모리를 해제해 주어야 한다.

본 프로그램은 명령어 수준의 병렬성을 소프트웨어 수준에서 구현하고 있다.

```
#define NUM_REG 32
#define PARALLEL_CNT 3
#define REG_PER_THREAD (NUM_REG /
PARALLEL_CNT)
#define BUBBLES 1
```

태스크들은 메모리 공간이 서로 독립되어 있으므로, 레지스터를 서로 겹치지 않게 사용한다면 명령어를 번갈아 가면서 배치해도 문제가 없다.

사용 가능한 레지스터는 전체적으로 줄어들지만, 사용하지 않는 것에 비해 CPI가 극적으로 상승하기 때문에 충분히 불편을 감수할 가치가 있다.

```
gppcu_push_instr(
    pp->MMAP_CMDOUT,
    pp->MMAP_DATOUT,
    GPPCU_ASSEMBLE_INSTRUCTION_C( COND_ALWAYS,
OPR_LDCI, 0, REGPIVOT, 0 )
);
for ( pcnt = 0; pcnt < PARALLEL_CNT; ++pcnt )
{
    gppcu_push_instr(
        pp->MMAP_CMDOUT,
        pp->MMAP_DATOUT,
        GPPCU_ASSEMBLE_INSTRUCTION_B(
            COND_ALWAYS,
            OPR_B_ADI,
            0,
            REGPIVOT + REG_PER_THREAD * ( pcnt
+ 1 ),
            REGPIVOT,
            pp->mtaskmem * ( pcnt + 1 )
        )
    );
}
```

가장 먼저 REGPIVOT에 태스크 오프셋을 할당한다. 이때, 하드웨어에게는 태스크 오프셋을 n 배만큼 뛰어넘을 것을 지시해 둔다.

```
gppcu_device_command( pp->MMAP_CMDOUT,
LPM_SZPERCYCLE, pp->mtaskmem *
PARALLEL_CNT );
gppcu_device_command(          pp->MMAP_CMDOUT,
LPM_NUMCYCLE, ( pp->mtaskcycle + PARALLEL_CNT
- 1 ) / PARALLEL_CNT );
```

전체 사이클은 그만큼 줄어들게 된다. 그러나 프로그램 크기는 병렬화 개수만큼 커지므로, 득실을 따져야 한다.

```
while ( lphead < lpend ) {
    // @todo. verify hardware stall generator
and remove this
    swk_gppcu_instr_t const instr = *lphead++;

    // @parallel instruction
    swk_gppcu_instr_t
parallel_instr[PARALLEL_CNT];

    const uint8_t opc = ( instr >> 23 ) &
0x1f;

    // don't repeat nop
    if ( opc == 0 )
    {
        gppcu_push_instr( pp->MMAP_CMDOUT, pp-
>MMAP_DATOUT, 0 );
        continue;
    }

    // Parallelize
    // gppcu_push_instr( pp->MMAP_CMDOUT, pp-
>MMAP_DATOUT, instr );

    for ( pcnt = 0; pcnt < PARALLEL_CNT;
pcnt++ )
    {
        swk_gppcu_instr_t* const pinstr =
parallel_instr + pcnt;
        const uint8_t toadd = ( pcnt ) *
REG_PER_THREAD;

        pinstr[0] = instr;
        if ( instr_userega( opc ) ) {
            pinstr[0] += toadd << 12;
        }
        if ( instr_useregb( opc ) ) {
            pinstr[0] += toadd << 0;
        }
        if ( instr_useregd( opc ) ) {
            pinstr[0] += toadd << 17;
        }
        gppcu_push_instr( pp->MMAP_CMDOUT, pp-
>MMAP_DATOUT, pinstr[0] );
        char buff[124];
        instr_to_string( buff, pinstr[0] );
        // printf( "putting instr %s\n",
buff );
    }

    if ( OPR_0_ITOF <= opc && opc <=
OPR_0_FSQRT )
    {
        gppcu_push_instr( pp->MMAP_CMDOUT, pp-
>MMAP_DATOUT, 0 );
    }
    for ( pcnt = 0; pcnt < BUBBLES; ++pcnt )
    {
        gppcu_push_instr( pp->MMAP_CMDOUT, pp-
>MMAP_DATOUT, 0 );
    }
}
```

이후 인스트럭션을 gppcu에 밀어 넣게 되는데, 이때 instruction이 register를 사용하는지 사용하지 않는지를 분석해서, 만약 사용한다면 레지스터 오프셋만큼 값을 더해 레지스터 공간을 분리해 준다. 설계가 어느정도 끝난 후에 이런 아이디어를 떠올려서 이렇게 구성되었지만, 처음부터 Banked register를 사용한다면 훨씬 깔끔한 구성이 되었을 듯해 아쉽다.

그렇게 병렬화 개수만큼 인스트럭션을 밀어 넣은 후, 버블을 삽입한다. 하드웨어의 설계가 미흡한 탓에 데이터 해저드가 빈번하게 발생하기 때문이다.

어플리케이션

실제 어플리케이션은 빈 공간에 빙글빙글 돌아가는 상자를 띄우는 프로그램이다.

mathc라고 하는 C 3D Math 라이브러리의 소스 코드를 임포트해서, 수월하게 구현할 수 있었다.

3D 렌더링 관련 로직은 하드웨어와 큰 관계가 없으므로, 이 보고서에서는 다루지 않는다.

중요한 부분은 여러 개의 버텍스(정점)를 gppcu에 밀어 넣고, 버텍스 셰이더를 수행한 뒤 다시 반환 받는 과정이다.

```
app_upload_vertices( &gppcu, &mesh );
app_upload_program( &gppcu );
```

최초 1회만 매시와 프로그램을 업로드하고,

```
while ( true )
{
    app_calc_object_constant(  &result,  &cam,
&mesh_inst );
```

이후에는 반복적으로 월드 행렬만 업데이트를 해준다. 이를 통해 매시의 회전이나 오프셋 등의 움직임을 즉각 반영할 수 있다.

고찰

CPU, GPU, 이런 복잡한 프로세서들에 관심이 많아 한 번 설계를 시도해 보았습니다. 다른 과목의 기말고사와 텀 프로젝트하고도 겹쳐 지난 학기 CPU를 만들 때처럼 그렇게 많은 시간을 투자하진 못했지만, 꾸준히 하드웨어 설계에 관심을 갖고 공부를 하면서 실력이 향상되었는지 이번 과제는 제법 짧은 시간 안에 괜찮은 결과를 얻어낼 수 있었습니다.

아무래도 멀티 사이클 FPU는 처음 계획에 없었는데, 급작스럽게 시스템에 끼워 넣다 보니 여러 문제가 속출했습니다. 첫 단추를 제대로 끼우는 게 정말 중요하다는 교훈을 이번에도 얻은 것 같습니다. 무엇보다도, 키보드를 잡기 전에 노트를 펼쳐 생각을 한 번 정리하는 게 어떤 시간 투자보다도 귀중하다는 걸 깨닫았습니다.

이번에 더듬더듬 파이프라인 설계를 어떻게 완성하고, 병렬 연산기를 설계하면서 조금 더 효율적인 아키텍처를 설계하고 싶은 욕심이 생겼습니다.

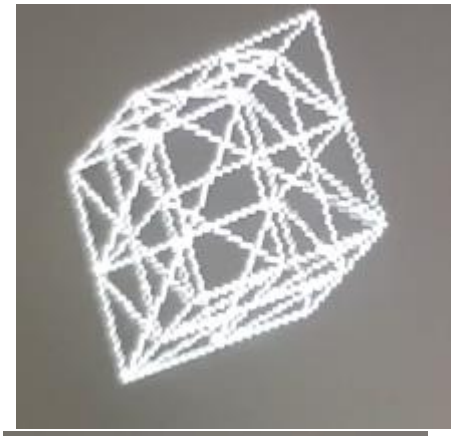
무엇보다도 NIOS의 Avalon BUS 인터페이스를 공부할 시간이 모자라, 결국 PIO를 이용해 주먹구구식으로 땜질한 설계가 된 것이 너무나 아쉽고, 고작 24개밖에 인스턴스화를 못했는데 로직 엘리먼트 숫자가 7만개를 넘어가 좀 더 대규모의 병렬 연산을 해내지 못한 게 아쉽습니다.

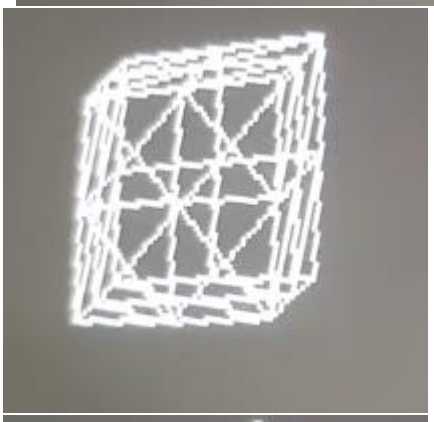
과제가 끝나고 제출할 때가 다 된 지금에야 깨달은 것인데, 멀티사이클 FPU의 본래 목적이 NIOS CPU의 커스텀 명령어를 지원하기 위해 설계된 만큼 이런 대량의 인스턴스화에는 적합하지가 않다는 점이었습니다.

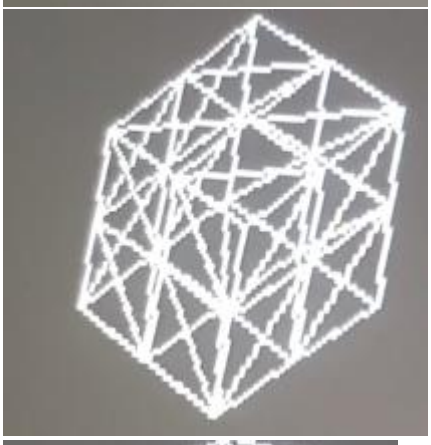
여러 모로 많은 교훈을 얻은 과제였고, 무엇보다도 C 프로그램과 베릴로그로 작성한 하드웨어의 상호작용을 하나씩 구현하는 게 굉장히 즐거웠습니다.

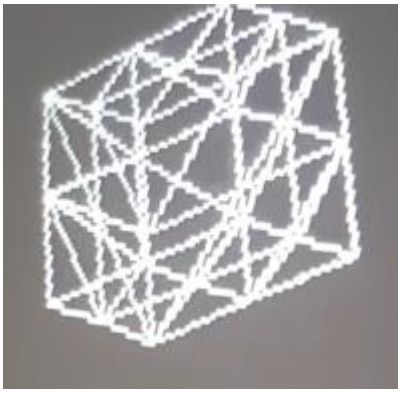
한 학기동안 고생 많으셨습니다!

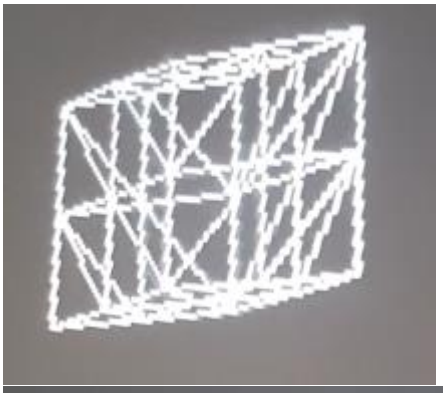
동작

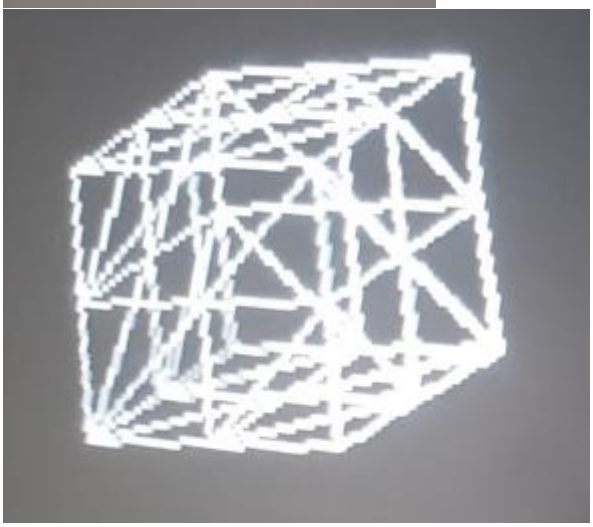












천천히 회전하는 상자를 띄운다.

동작에 대한 자세한 설명과 실 동작 영상이 담긴 유튜브 영상은 아래 링크에 있습니다.

https://youtu.be/9IIGGn2loE8