Introduction to High-Level Synthesis — Part II: Practice

- Introduction to High-Level Synthesis — Part II (ภาคปฏิบัติ)
  - สารบัญ Part II
  - บทที่ 7 — การติดตั้ง Bambu HLS
  - บทที่ 8 — Lab 1: Simple Adder
  - บทที่ 9 — Lab 2: FIR Filter
  - บทที่ 10 — Lab 3: Matrix Multiplication
  - บทที่ 11 — Lab 4: Dataflow Streaming
  - บทที่ 12 — Verification
  - บทที่ 13 — Integration กับ OpenROAD
  - บทที่ 14 — Best Practices และ Pitfalls
  - ภาคผนวก — Cheat-Sheet
  - แบบฝึกหัดรวม (Final Project)

Introduction to High-Level Synthesis — Part II (ภาคปฏิบัติ)

> ต่อจาก Part I (ทฤษฎี) — เอกสารฉบับนี้เน้นการลงมือทำจริงกับ **Bambu HLS** และผสานเข้ากับ flow Open-source (Yosys / OpenROAD / Icarus / COCOTB)

สารบัญ Part II

7. การติดตั้ง Bambu HLS
8. Lab 1 — Simple Adder (Hello World)
9. Lab 2 — FIR Filter with Pipelining
10. Lab 3 — Matrix Multiplication
11. Lab 4 — Dataflow Streaming
12. Verification ด้วย Icarus และ COCOTB
13. Integration กับ OpenROAD (IHP SG13G2 / SKY130)
14. Best Practices และ Pitfalls
15. ภาคผนวก: Cheat-Sheet

บทที่ 7 — การติดตั้ง Bambu HLS

7.1 ทำไมเลือก Bambu?

Bambu เป็น HLS tool open-source ที่:

- เขียนด้วย C++ (ดูแลโดย PoliMi)
- รับ C/C++ → ส่ง Verilog/VHDL

- **ออกแบบมาเพื่อ ASIC flow** (ไม่ใช่แค่ FPGA)
- ใช้ร่วมกับ **Yosys** สำหรับ logic synthesis
- รองรับ multiple backends: Vivado (Xilinx FPGA), OpenROAD (ASIC), Catapult, custom
- ไม่ผูกกับ vendor

7.2 ความต้องการของระบบ

Resource	Minimum	Recommended
OS	Ubuntu 20.04+ / WSL2 / macOS	Ubuntu 22.04
RAM	8 GB	16 GB+
Disk	5 GB	10 GB
Compiler	gcc/g++ 9+	gcc 11
Build tools	make, autoconf, libtool, python3	+ clang-tidy, boost

7.3 การติดตั้งบน Ubuntu / WSL2

```
# 1. ติดตั้ง dependencies
sudo apt-get update
sudo apt-get install -y \
    build-essential gcc g++ \
    autoconf automake libtool \
    python3 python3-dev \
    libboost-all-dev \
    libtcl-dev tk-dev \
    libreadline-dev \
    zlib1g-dev \
    git cmake

# 2. Clone source
cd ~
git clone https://github.com/ferrandi/PandA-bambu.git
cd PandA-bambu

# 3. Build
make -j$(nproc)

# 4. ทดสอบ
./bambu --version
```

7.4 การติดตั้งผ่าน Docker (แนะนำสำหรับ class)

```
docker pull christopherzimmerman/bambu:latest
docker run -it --rm -v $(pwd):/work christopherzimmerman/bambu:latest
```

7.5 โครงสร้าง Output ของ Bambu

เมื่อรัน Bambu จะได้ไฟล์ดังนี้:

```
out/
├── HLS_output/
│   ├── verilog/
│   │   ├── dut.v                # top-level module
│   │   ├── dut_datapath.v       # datapath
│   │   ├── dut_controller.v     # FSM controller
│   │   ├── dut_slow_RAM_*       # inferred memory
│   │   └── dut_tanhfxp_sigmoid_table.dat   # LUT
│   ├── simulation/
│   │   ├── test_dut.v           # Verilog testbench
│   │   ├── test_dut.log         # simulation log
│   │   └── results.txt          # PASS/FAIL
│   ├── HLS_summary.json
│   ├── HLS_synthesis_report.txt  # resource report
│   └── bambu_log.txt
```

7.6 คำสั่งพื้นฐาน

```
# แปลง C → Verilog (ไม่ simulate)
bambu \
    --top-fname=dut \
    file.c

# ระบุ target clock
bambu --top-fname=dut --clock-period=10 file.c

# Run co-simulation (C testbench + Verilog via Icarus)
bambu --top-fname=dut --simulate --simulator=VERILATOR file.c

# Generate for OpenROAD backend
bambu --top-fname=dut --backend=OpenROAD file.c

# ดู resource report
cat out/HLS_output/HLS_synthesis_report.txt
```

บทที่ 8 — Lab 1: Simple Adder

เป้าหมาย: เรียนรู้ Bambu HLS เบื้องต้น — ดูว่าโค้ด C ธรรมดากลายเป็น Verilog อย่างไร

8.1 ไฟล์ที่ต้องสร้าง

```
lab1-adder/
├── adder.c          # HLS source
├── adder.h          # header
└── run.sh           # script
```

8.2 `adder.c`

```c
// adder.c
#include "adder.h"

void adder(int a, int b, int* c) {
    *c = a + b;
}

int main() {
    int a = 5, b = 7, c;
    adder(a, b, &c);
    if (c != 12) {
        printf("FAIL: expected 12, got %d\n", c);
        return 1;
    }

    a = -100; b = 200;
    adder(a, b, &c);
    if (c != 100) {
        printf("FAIL: expected 100, got %d\n", c);
        return 1;
    }

    printf("PASS\n");
    return 0;
}
```

8.3 `adder.h`

```c
#ifndef ADDER_H
#define ADDER_H

#include <stdio.h>

void adder(int a, int b, int* c);

#endif
```

8.4 `run.sh`

```bash
#!/usr/bin/env bash
set -euo pipefail

LAB_DIR="$(cd "$(dirname "$0")" && pwd)"
cd "$LAB_DIR"

echo "========================================"
echo "Lab 1: Simple Adder"
echo "========================================"

# Step 1: HLS synthesis (C -> Verilog)
echo "[1/3] Running Bambu HLS..."
bambu \
    --top-fname=adder \
    -O2 \
    --clock-period=10 \
    --generate-tb=testbench=main \
    --simulate \
    --simulator=VERILATOR \
    --print-dot \
    adder.c

# Step 2: Inspect output
echo ""
echo "[2/3] Generated files:"
ls -la HLS_output/verilog/

# Step 3: View resource report
echo ""
echo "[3/3] Resource report:"
cat HLS_output/HLS_synthesis_report.txt
```

8.5 การรัน

```
cd lab1-adder
chmod +x run.sh
./run.sh
```

8.6 ผลลัพธ์ที่คาดหวัง

```
==========================================
Lab 1: Simple Adder
==========================================
[1/3] Running Bambu HLS...
================ Bambu Log ================
...
[SUCCESS] Co-simulation finished: PASSED
==========================================
[2/3] Generated files:
HLS_output/verilog/adder.v
HLS_output/verilog/adder_datapath.v
HLS_output/verilog/adder_controller.v
[3/3] Resource report:
==========================================
==  Bambu HLS Resource Report
==========================================
Module: adder
  Resource utilization:
    LUTs:    ~10
    FFs:     ~50
    DSPs:    0
    BRAMs:   0
    Latency: 1 cycle
    Interval: 1 cycle
```

8.7 Verilog ที่ได้ — ดูด้วยตา

```
cat HLS_output/verilog/adder.v
```

จะเห็น:

```
module adder(
    input  clock,
    input  reset,
    input  start_port,
    output done_port,
    input  [31:0] a,
    input  [31:0] b,
    output [31:0] c,
    // ... control signals
);
    // FSM
    reg [1:0] _tmp_1;
    always @(posedge clock or posedge reset) begin
        if (reset) _tmp_1 <= 2'd0;
        else _tmp_1 <= /* next state */;
    end
    // datapath
    assign c = a + b;
endmodule
```

8.8 แบบฝึกหัด

1. เปลี่ยน `int` เป็น `char` (8-bit) — ดู resource report เปลี่ยนอย่างไร
2. เพิ่ม `* 3` หลังการบวก — ดูว่าเกิด DSP ในรายงานหรือไม่
3. ลอง `--clock-period=5` (เร็วขึ้น) — Bambu อาจเพิ่ม pipeline stage

บทที่ 9 — Lab 2: FIR Filter

เป้าหมาย: เรียนรู้การใช้ `#pragma HLS PIPELINE` เพื่อเพิ่ม throughput

9.1 ทฤษฎี FIR

FIR (Finite Impulse Response) filter:

$$y[n] = \sum_{k=0}^{N-1} h[k] \cdot x[n-k]$$

ใช้ shift register + MAC (multiply-accumulate)

9.2 `fir.c`

```c
// fir.c
// 8-tap FIR filter, 16-bit signed input
#include "fir.h"

#define N_TAPS 8

// Filter coefficients (low-pass, normalized)
const int coeffs[N_TAPS] = {
    100, 250, 500, 700, 700, 500, 250, 100
};

void fir(int x, int* y) {
    static int shift_reg[N_TAPS] = {0};

#pragma HLS PIPELINE II=1
    // Shift register
    for (int i = N_TAPS - 1; i > 0; i--) {
        shift_reg[i] = shift_reg[i-1];
    }
    shift_reg[0] = x;

    // MAC
    int acc = 0;
    for (int k = 0; k < N_TAPS; k++) {
        acc += shift_reg[k] * coeffs[k];
    }
    *y = acc;
}

int main() {
    int x, y;

    // Test 1: impulse response
    // Input: 1, 0, 0, ..., 0 → output: coeffs[0], coeffs[1], ...
    x = 1024; fir(x, &y);  // y should ≈ 100 * 1024
    if (y < 90000 || y > 110000) {
        printf("FAIL T1: got %d\n", y); return 1;
    }

    for (int i = 0; i < N_TAPS; i++) {
        x = 0; fir(x, &y);
    }
    // After N_TAPS zeros, y should be ~0

    // Test 2: step response
    for (int i = 0; i < 20; i++) {
        x = 1024; fir(x, &y);
    }
    // Should be ~ sum(coeffs) * 1024
```

```
    printf("PASS\n");
    return 0;
}
```

9.3 `fir.h`

```
#ifndef FIR_H
#define FIR_H

#include <stdio.h>

void fir(int x, int* y);

#endif
```

9.4 `run.sh`

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")"

echo "=== Lab 2: FIR Filter with Pipelining ==="

# Baseline (no directive)
echo ""
echo "--- A. Baseline (no PIPELINE) ---"
bambu --top-fname=fir --clock-period=10 --simulate fir.c \
    2>&1 | tail -20

# With PIPELINE II=1
echo ""
echo "--- B. With PIPELINE II=1 ---"
bambu --top-fname=fir --clock-period=10 --simulate fir.c \
    --pragma="HLS PIPELINE II=1" \
    2>&1 | tail -20

# Resource comparison
echo ""
echo "--- C. Resource Report ---"
cat HLS_output/HLS_synthesis_report.txt
```

9.5 ผลลัพธ์ที่คาดหวัง

Configuration	Latency	II	DSPs	LUTs
No PIPELINE	~50 cycles	50	1	~200
PIPELINE II=1	~10 cycles	1	8	~800

สังเกต: - PIPELINE II=1 → DSP เพิ่มเป็น 8 (เพราะ unroll implicit) - Throughput เพิ่มขึ้น 50× - Area เพิ่มขึ้น 4× — **trade-off!**

9.6 แบบฝึกหัด

1. ลอง `PIPELINE II=2` — area vs throughput เปลี่ยนอย่างไร
2. เพิ่ม N_TAPS เป็น 16, 32 — observe scaling
3. ใช้ `ap_int<16>` แทน `int` (Bambu ใช้ `ac_int<16,true>`) — observe DSP/LUT

บทที่ 10 — Lab 3: Matrix Multiplication

เป้าหมาย: เรียนรู้ ARRAY_PARTITION + UNROLL เพื่อจัดการ memory bottleneck

10.1 โค้ด

```c
// matmul.c
#include "matmul.h"

#define N 8

void matmul(int A[N][N], int B[N][N], int C[N][N]) {
#pragma HLS ARRAY_PARTITION variable=A complete dim=2
#pragma HLS ARRAY_PARTITION variable=B complete dim=1
#pragma HLS ARRAY_PARTITION variable=C complete dim=2

    for (int i = 0; i < N; i++) {
#pragma HLS PIPELINE II=1
        for (int j = 0; j < N; j++) {
            int sum = 0;
            for (int k = 0; k < N; k++) {
#pragma HLS UNROLL factor=4
                sum += A[i][k] * B[k][j];
            }
            C[i][j] = sum;
        }
    }
}

int main() {
    static int A[N][N], B[N][N], C[N][N];

    // Initialize
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            A[i][j] = i + j;
            B[i][j] = i * 2 + j;
        }

    matmul(A, B, C);

    // Reference computation
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            int ref = 0;
            for (int k = 0; k < N; k++) {
                ref += A[i][k] * B[k][j];
            }
            if (C[i][j] != ref) {
                printf("FAIL at [%d][%d]: %d != %d\n", i, j, C[i][j], ref);
                return 1;
            }
        }
    }
    printf("PASS\n");
```

```
    return 0;
}
```

10.2 `matmul.h`

```
#ifndef MATMUL_H
#define MATMUL_H

#include <stdio.h>

#define N 8
void matmul(int A[N][N], int B[N][N], int C[N][N]);

#endif
```

10.3 `run.sh`

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")"
bambu --top-fname=matmul --clock-period=10 --simulate matmul.c
echo "--- Resource Report ---"
cat HLS_output/HLS_synthesis_report.txt
```

10.4 การวิเคราะห์

Optimization	Effect
`ARRAY_PARTITION A dim=2`	แยก row → เข้าถึง A[i][0..7] พร้อมกัน
`ARRAY_PARTITION B dim=1`	แยก column → เข้าถึง B[0..7][j] พร้อมกัน
`UNROLL factor=4`	4× parallel MAC
`PIPELINE II=1`	inner loop throughput สูงสุด

10.5 Trade-off Visualization

```
Throughput (II=1, full unroll):
  → 1 MAC / cycle
  → 64 cycles สำหรับ N=8

Latency (no optimization):
  → 8x8x8 = 512 cycles แบบ sequential
```

10.6 แบบฝึกหัด

1. เปลี่ยน N เป็น 4, 16 — observe synthesis time
2. ลอง `BLOCK factor=4` แทน `complete` — compare resource
3. ลอง `DATAFLOW` ระหว่าง for loop

บทที่ 11 — Lab 4: Dataflow Streaming

เป้าหมาย: เรียนรู้ task-level parallelism ด้วย DATAFLOW และ hls::stream

11.1 สถาปัตยกรรม

```
Input Stream → [Stage 1: +1] → [Stage 2: ×2] → [Stage 3: -5] → Output Stream
               FIFO1              FIFO2              FIFO3
```

11.2 `dataflow.cpp`

```
// dataflow.cpp
#include "dataflow.h"

void stage1(hls::stream<int>& in, hls::stream<int>& out) {
#pragma HLS INLINE off
    int x = in.read();
    out.write(x + 1);
}

void stage2(hls::stream<int>& in, hls::stream<int>& out) {
#pragma HLS INLINE off
    int x = in.read();
    out.write(x * 2);
}

void stage3(hls::stream<int>& in, hls::stream<int>& out) {
#pragma HLS INLINE off
    int x = in.read();
    out.write(x - 5);
}

void pipeline(hls::stream<int>& in, hls::stream<int>& out) {
#pragma HLS DATAFLOW
    hls::stream<int> mid1, mid2;
#pragma HLS STREAM variable=mid1 depth=4
#pragma HLS STREAM variable=mid2 depth=4

    stage1(in, mid1);
    stage2(mid1, mid2);
    stage3(mid2, out);
}

int main() {
    hls::stream<int> in, out;

    // Drive 10 samples
    for (int i = 0; i < 10; i++) {
        in.write(i);
    }

    // Run pipeline
    pipeline(in, out);

    // Check: out[i] should equal (i+1)*2 - 5
    for (int i = 0; i < 10; i++) {
        int got = out.read();
        int expected = (i + 1) * 2 - 5;
        if (got != expected) {
            printf("FAIL at i=%d: got %d, expected %d\n", i, got, expected);
```

```
            return 1;
        }
    }
    printf("PASS\n");
    return 0;
}
```

11.3 `dataflow.h`

```
#ifndef DATAFLOW_H
#define DATAFLOW_H

#include <hls_stream.h>
#include <stdio.h>

void pipeline(hls::stream<int>& in, hls::stream<int>& out);

#endif
```

11.4 `run.sh`

```
#!/usr/bin/env bash
set -euo pipefail
cd "$(dirname "$0")"
bambu --top-fname=pipeline --clock-period=10 --simulate dataflow.cpp
cat HLS_output/HLS_synthesis_report.txt
```

11.5 Pipeline Behavior

```
Clock:     0   1   2   3   4   5   6   7
Stage1:   [a] [b] [c] [d] ...
Stage2:       [a] [b] [c] ...
Stage3:           [a] [b] ...
output:               [a] [b] ...
```

- **Latency** ของ data point แรก: 3 cycles (fill pipeline)
- **Throughput**: 1 sample/cycle (หลังจาก pipeline fill)

11.6 แบบฝึกหัด

1. เพิ่ม stage 4: saturate ที่ 100
2. เปลี่ยน FIFO depth เป็น 1, 8 — observe area
3. ใช้ `ap_axis<>` แทน `int` เพื่อสร้าง AXI-Stream interface

บทที่ 12 — Verification

12.1 ระดับชั้นของ Verification ใน HLS

ระดับ	สิ่งที่ตรวจ	เครื่องมือ
1. C algorithmic	logic ของ algorithm	gcc run, gdb
2. C/RTL co-sim	functional ของ generated RTL	Bambu –simulate + Icarus/Verilator
3. RTL standalone	standalone Verilog	Icarus, Verilator
4. Gate-level (post-synth)	หลัง Yosys/OpenROAD	Icarus + SDF
5. On hardware	bitstream โหลดลง FPGA	Vivado, LiteX

12.2 Bambu Co-Simulation แบบละเอียด

```
# ใช้ Verilator (เร็วกว่า Icarus)
bambu --top-fname=dut --simulate --simulator=VERILATOR file.c

# ใช้ Icarus
bambu --top-fname=dut --simulate --simulator=ICARUS file.c

# ใช้ ModelSim (commercial)
bambu --top-fname=dut --simulate --simulator=MODELSIM file.c
```

12.3 ดู Generated Testbench

```
cat HLS_output/simulation/test_dut.v
```

ไฟล์นี้ถูก generate อัตโนมัติ — คุณไม่ต้องเขียน HDL testbench

12.4 ใช้ COCOTB สำหรับ Advanced Verification

COCOTB (Python-based testbench) ใช้กับ Verilog/VHDL ได้ — เหมาะกับ complex test:

`cocotb_test.py`

```python
import cocotb
from cocotb.triggers import RisingEdge, Timer
from cocotb.clock import Clock

@cocotb.test()
async def test_fir(dut):
    # Clock 100 MHz
    cocotb.start_soon(Clock(dut.clock, 10, units="ns").start())

    # Reset
    dut.reset.value = 1
    dut.start_port.value = 0
    await RisingEdge(dut.clock)
    await RisingEdge(dut.clock)
    dut.reset.value = 0
    await RisingEdge(dut.clock)

    # Send 10 samples
    expected = [100, 350, 850, 1550, 2250, 2950, 3450, 3450, 2950, 2450]
    for i, exp in enumerate(expected):
        dut.a.value = 1024 if i == 0 else 0
        dut.b.value = 0  # for simpler test
        dut.start_port.value = 1
        await RisingEdge(dut.clock)
        dut.start_port.value = 0
        # Wait done
        for _ in range(20):
            await RisingEdge(dut.clock)
            if dut.done_port.value == 1:
                break
        cocotb.log.info(f"sample {i}: y={int(dut.y.value)}")
```

Run:

```bash
cd HLS_output/verilog
cocotb-run --make
```

12.5 Self-Checking Testbench ใน C

```c
int main() {
    int errors = 0;
    for (int t = 0; t < 100; t++) {
        int a = rand() % 1000;
        int b = rand() % 1000;
        int c;
        dut(a, b, &c);
        if (c != a + b) {
            printf("FAIL: %d + %d = %d (got %d)\n", a, b, a+b, c);
            errors++;
        }
    }
    return errors ? 1 : 0;  // Bambu uses return value as PASS/FAIL
}
```

> **เคล็ดลับ:** ใช้ return value 0 = PASS, ไม่ใช่ 0 = FAIL เพื่อให้ Bambu self-check ได้

บทที่ 13 — Integration กับ OpenROAD

13.1 Flow ภาพรวม

```
C source ──Bambu──▶ Verilog ──Yosys──▶ Netlist ──OpenROAD──▶ GDSII
                      |                   |
                      ▼                   ▼
                Co-simulation        Floorplan/Place/Route
                  (Icarus)           (IHP SG13G2 / SKY130)
```

13.2 Step 1: Generate Verilog ด้วย Bambu

```bash
bambu \
    --top-fname=matmul \
    --clock-period=10 \
    --backend=OpenROAD \
    --memory-allocation-strategy=BAMBU \
    matmul.c
```

จะได้ไฟล์: - `HLS_output/verilog/matmul.v` — top module - `HLS_output/verilog/matmul_datapath.v` — datapath - `HLS_output/verilog/matmul_controller.v` — FSM - `HLS_output/verilog/matmul_slow_RAM_*.v` — inferred BRAMs

13.3 Step 2: Synthesize ด้วย Yosys

สร้าง `synth.ys`:

```
# synth.ys
read_verilog HLS_output/verilog/matmul.v
read_verilog -sv HLS_output/verilog/matmul_datapath.v
read_verilog -sv HLS_output/verilog/matmul_controller.v
read_verilog -sv HLS_output/verilog/matmul_slow_RAM_*.v

hierarchy -check -top matmul

# PDK-specific liberty file
read_liberty -lib /path/to/ihp-sg13g2_stdcell.lib

synth -top matmul
abc -liberty /path/to/ihp-sg13g2_stdcell.lib
write_verilog synth/matmul_synth.v
write_json synth/matmul.json
```

Run:

```
yosys -s synth.ys
```

13.4 Step 3: OpenROAD Flow

`run_openroad.tcl`:

```tcl
# run_openroad.tcl
set design matmul
set top_module matmul
set pdk_dir /path/to/IHP-Open-PDK

# Read liberty
read_liberty ${pdk_dir}/libs.ref/sg13g2_stdcell/lib/sg13g2_stdcell.lib

# Read netlist
read_verilog synth/matmul_synth.v
link_design $top_module

# Constraints
create_clock -name clk -period 10 [get_ports clk]
set_input_delay 2 -clock clk [all_inputs]
set_output_delay 2 -clock clk [all_outputs]

# Floorplan
initialize_floorplan \
    -die_area "0 0 200 200" \
    -core_area "10 10 190 190" \
    -site unithv

# Place
global_placement -density 0.6
detailed_placement

# CTS
clock_tree_synthesis -root_buf sg13g2_buf_8

# Route
global_routing
detailed_routing

# Reports
report_timing > reports/timing.rpt
report_area > reports/area.rpt
write_def synth/matmul.def
write_gdsii synth/matmul.gds
```

13.5 LibreLane Flow (อัตโนมัติ)

สร้าง `config.json`:

```json
{
  "DESIGN_NAME": "matmul",
  "VERILOG_FILES": [
    "HLS_output/verilog/matmul.v",
    "HLS_output/verilog/matmul_datapath.v",
    "HLS_output/verilog/matmul_controller.v",
    "HLS_output/verilog/matmul_slow_RAM_*.v"
  ],
  "CLOCK_PERIOD": 10,
  "CLOCK_PORT": "clock",
  "PDK": "ihp-sg13g2",
  "FP_CORE_UTIL": 50,
  "FP_SIZING": "absolute",
  "DIE_AREA": [0, 0, 200, 200],
  "CORE_AREA": [10, 10, 190, 190]
}
```

```bash
librelane --config config.json --tag hls-run .
```

13.6 SKY130 Variant

```json
{
  "DESIGN_NAME": "matmul",
  "VERILOG_FILES": [...],
  "CLOCK_PERIOD": 10,
  "PDK": "sky130A",
  "STD_CELL_LIBRARY": "sky130_fd_sc_hd"
}
```

13.7 ผลลัพธ์ที่คาดหวัง (Post-Route)

Metric	Lab 1 (Adder)	Lab 2 (FIR)	Lab 3 (MatMul)	Lab 4 (Dataflow)
Area (μm²) @ IHP SG13G2	~500	~2,000	~8,000	~1,200
Fmax (MHz)	200	150	100	180
Latency (cycles)	1	~10	~80	~5
Throughput	1/cyc	1/cyc	1/cyc	1/cyc
Power (μW)	50	500	5,000	200

บทที่ 14 — Best Practices และ Pitfalls

14.1 Top 10 Pitfalls

1. Loop bound ไม่คงที่

```
// ❌ Bad
for (int i = 0; i < N; i++)  // N จาก input
// ✅ Good (ใช้ #pragma)
#pragma HLS LOOP_TRIPCOUNT min=8 max=64
for (int i = 0; i < N; i++)
```

2. Pointer aliasing

```
// ❌ HLS assume alias
void foo(int* a, int* b) { *a = *b + 1; }
// ✅ ใช้ restrict
void foo(int* restrict a, int* restrict b) { *a = *b + 1; }
```

3. Array เข้าถึง conflict

```
// ❌ ต้อง read แล้ว write ที่เดียวกัน → conflict
buf[i] = buf[i-1] + 1;
// ✅ ใช้ shift register หรือ partition
```

4. **Resource หมด (II ไม่ลดลง)**

```
Warning: Cannot schedule with II=1, minimum II = 4
```

→ เพิ่ม resource (UNROLL, partition) หรือ ยอมรับ II ที่สูงกว่า

5. **Function ไม่ถูก inline**
   - เพิ่ม `INLINE` directive
   - หรือ เปลี่ยนเป็น macro
6. **Memory bottleneck**
   - ใช้ `ARRAY_PARTITION` หรือ local variable
   - ใช้ streaming interface
7. **Floating-point vs Fixed-point**
   - float ใช้ resource มาก → ใช้ fixed-point (`ac_fixed`, `ap_fixed`)
   - tradeoff: range/precision
8. **Static vs Global**
   - `static` ใน function = เก็บ state (เช่น shift_reg) → OK
   - `static` นอก function = global → ระวัง initialization
9. **Printf ใน synthesized function**
   - Bambu/Vitis จะ error
   - เอา printf ออกจาก top function
10. **ไม่ self-checking testbench**
    - return 0 = PASS, !=0 = FAIL
    - Bambu จะใช้ตรวจ co-sim

14.2 Debugging Workflow

```
1. Compile C with gcc → ตรวจ algorithm ก่อน
2. Run Bambu --simulate → ตรวจ RTL functional
3. ดู HLS_synthesis_report.txt → ตรวจ resource
4. ดู generated Verilog → ตรวจ micro-architecture
5. ใช้ waveform (--generate-vcd) → debug cycle-by-cycle
```

Generate waveform:

```
bambu --top-fname=dut --simulate --generate-vcd --vcd-fname=dut.vcd dut.c
gtkwave dut.vcd
```

14.3 Performance Tuning Recipe

```
// เริ่มจาก baseline
void kernel(...) { ... }

// Level 1: PIPELINE บน inner loop
#pragma HLS PIPELINE II=1
for (...) ...

// Level 2: ARRAY_PARTITION ถ้า memory bottleneck
#pragma HLS ARRAY_PARTITION variable=arr complete

// Level 3: UNROLL บน inner-most
#pragma HLS UNROLL factor=N

// Level 4: DATAFLOW ระหว่าง function
#pragma HLS DATAFLOW

// Level 5: เปลี่ยน algorithm (loop interchange, tiling)
```

14.4 การเลือก Optimization Level

Goal	Strategy
Minimize area	ไม่ PIPELINE, share resource, เล็ก type
Maximize throughput	PIPELINE II=1 + UNROLL + PARTITION
Balance	PIPELINE II=2–4, partial UNROLL
Minimize latency	DATAFLOW + parallel stages

ภาคผนวก — Cheat-Sheet

A.1 Bambu Command-Line Quick Reference

```
# Basic
bambu --top-fname=fn file.c

# Common flags
--top-fname=NAME              # top function name
--clock-period=NS             # target clock period (ns)
-O0, -O1, -O2, -O3            # optimization level
--device=xc7a100tcsg324-1     # target FPGA
--backend=OpenROAD            # ASIC backend

# Directives
--pragma="HLS PIPELINE II=1"
--pragma="HLS UNROLL factor=4"
--pragma="HLS ARRAY_PARTITION variable=arr complete"

# Simulation
--simulate                    # run C/RTL co-simulation
--simulator=VERILATOR         # verilator
--simulator=ICARUS            # icarus
--generate-vcd                # output waveform
--vcd-fname=trace.vcd

# Output
--output-dir=out              # output directory
--print-dot                   # print CDFG as .dot

# Memory
--memory-allocation-strategy=BAMBU
--bram-load-bytes=4
```

A.2 Vitis HLS ⟷ Bambu Directive Mapping

Concept	Vitis HLS	Bambu
Pipeline loop	`#pragma HLS PIPELINE`	เหมือนกัน
Unroll	`#pragma HLS UNROLL`	เหมือนกัน
Partition	`#pragma HLS ARRAY_PARTITION`	เหมือนกัน
Dataflow	`#pragma HLS DATAFLOW`	เหมือนกัน
Interface	`#pragma HLS INTERFACE`	Bambu auto-infer
Arbitrary int	`ap_int<N>`	`ac_int<N,true/false>`
Stream	`hls::stream<T>`	`hls::stream<T>`
Fixed point	`ap_fixed<Q,N>`	`ac_fixed<Q,N,...>`

A.3 Common File Templates

Top function template:

```
#include "types.h"  // ap_int / ac_int
#include "hls_stream.h"  // hls::stream

void dut_top(
    // input data
    int in_data,
    // output
    int* out_data
) {
    // static for state
    static int state_reg = 0;

#pragma HLS PIPELINE II=1
    // compute
    *out_data = state_reg + in_data;
    state_reg = in_data;
}

int main() {
    int in, out;
    in = 5;  dut_top(in, &out);
    if (out != 0) return 1;  // initial state

    in = 10; dut_top(in, &out);
    if (out != 5) return 1;

    in = 20; dut_top(in, &out);
    if (out != 10) return 1;

    return 0;  // PASS
}
```

Stream-based template:

```
#include "hls_stream.h"
void dut_stream(
    hls::stream<int>& in,
    hls::stream<int>& out
) {
#pragma HLS PIPELINE II=1
    int x = in.read();
    out.write(x * 2);
}
```

A.4 Bambu Exit Codes

Code	Meaning
0	Success (PASS)
1	Synthesis failed
2	Co-simulation mismatch (FAIL)
3	Timeout

A.5 Useful Bambu Log Sections

```
# Resource utilization
grep -A 20 "Resource utilization" out/HLS_output/HLS_synthesis_report.txt

# Latency
grep "Latency\|II " out/HLS_output/HLS_synthesis_report.txt

# Critical path
grep "Critical" out/HLS_output/HLS_synthesis_report.txt

# Errors
grep -i "error" out/bambu_log.txt
```

A.6 IHP SG13G2 + HLS เฉพาะ

Issue	Solution
HLS uses too much area	ลด UNROLL, ใช้ fixed-point
BRAM ไม่พอ	ใช้ distributed RAM, ลด array size
ไม่มี hardware multiplier ใน low-density	ใช้ shift-and-add (Bambu auto)
Clock ไม่ถึง target	เพิ่ม pipeline stage, ลบ DATAFLOW

A.7 References

1. **Bambu Documentation**: https://panda.dei.polimi.it/?page_id=31
2. **PandA Framework**: https://github.com/ferrandi/PandA-bambu
3. **Xilinx UG902** (Vitis HLS User Guide): https://docs.xilinx.com
4. **OpenROAD Flow**: https://openroad.readthedocs.io
5. **IHP SG13G2 PDK**: https://github.com/IHP-GmbH/IHP-Open-PDK

6. **SKY130 PDK**: https://skywater-pdk.readthedocs.io

7. **COCOTB**: https://docs.cocotb.org

8. **"High-Level Synthesis: From Algorithm to Digital Circuit"** — Philippe Coussy et al., Springer 2008

แบบฝึกหัดรวม (Final Project)

โจทย์: ออกแบบ **Sobel Edge Detector** ด้วย HLS

1. Input: 64×64 grayscale image (8-bit pixel)
2. Output: 64×64 edge-magnitude image
3. Filter kernel:
    - Gx = [[-1, 0, 1], [-2, 0, 2], [-1, 0, 1]]
    - Gy = [[-1, -2, -1], [0, 0, 0], [1, 2, 1]]
    - magnitude = sqrt($Gx^2 + Gy^2$) (หรือ approximation |Gx| + |Gy|)

งานที่ต้องส่ง: 1. Source code C/C++ (`sobel.c`) 2. Bambu HLS report 3. Generated Verilog 4. Resource utilization (LUT, FF, DSP, BRAM) 5. (Bonus) นำไปสังเคราะห์ผ่าน OpenROAD ด้วย IHP SG13G2 หรือ SKY130

Hint: - ใช้ `static` array สำหรับ line buffer - ใช้ PIPELINE + ARRAY_PARTITION - ทดลองทั้ง 2 precision: int vs fixed-point - เปรียบเทียบ resource ก่อน/หลัง optimization

---

จบหลักสูตร

> หากมีคำถามเพิ่มเติมหรือต้องการให้ทำ slide PPTX, สร้างเพิ่ม Lab เฉพาะทาง (เช่น HLS สำหรับ ML accelerator, RISC-V pipeline), หรือต้องการเวอร์ชัน PDF/PPTX — แจ้งได้เลยครับ