Lecture 5: Outline

- **SoC design flow**
  - Iterative design convergence

- **Requirement & performance analysis**
  - Algorithm analysis
  - Component-level estimation
  - System-level estimation
SoC Design Flow

Design Convergence and Verification Loop

Product Validation Loop

Start
- MRD
- PRD

Modify Model?
- Yes
- No

Map, Model & Simulate in SPW or Matlab or C or C++

Mapping to Platform or Components Complete?
- Yes
- No

Analyze results
- Yes
- No

Metric Met?
- Yes
- No

Freeze Architecture

MRD Met?
- Yes
- No

Done

Analyze results
- Yes
- No

Functionality Met?
- Yes
- No

System BOM Costs Met?
- Yes
- No

Power Req. Met?
- Yes
- No

Schedule Req. Met?
- Yes
- No

Platform Req. Met?
- Yes
- No

Return

Design Convergence

Front End Design
- Reduced convergence time due to minimal data
- Convergence time increases due to more design data

Implementation
- Reduced convergence time due to reduced solution space
- Convergence time increases due to transition phase
- Reduced convergence time due to reduced solution space

Rapid Exploration
- Design Converges

Rapid Traversal
Performance of a System

- **Depends on many factors**
  - System design (algorithms and data structures)
  - Implementation (code)
  - Execution platform architecture
  - The workload to which it is subjected
  - The metric used in the evaluation

- **Interactions between these factors**

Performance Analysis

- **Predictability**
  - w.c. guarantee
  - w.c. performance

- **Variation of actual execution times**

- **Simulation (dynamic) vs. analysis (static)**
  - Tightness of bounds, under- vs. over-estimation
Evaluating a Design

- **Algorithm level**
  - Analysis of complexity
  - Identify bottlenecks, evaluate tradeoffs

- **Component level**
  - Software partition
    - Profiling, measurement
    - Simulation, tracing
    - Power/performance/... estimation
  - Hardware partition
    - Modeling & estimation

- **System level**
  - Virtual platform prototyping
  - Physical prototyping

Algorithm Analysis

- **Theoretical complexity analysis**
  - $O()$ notation ("order-of")
  - Example: Sorting
    - "Bubble" sort
    - Merge sort
  - Example: Fourier transform
    - Discrete Fourier transform
    - Fast Fourier transform

- **Analysis of code**
  - Example: BCH* encoding
    - Code in C
    - Find the number of XOR and AND operations performed in the loop as a function of $k$
    - Assume length is 1024, and in any bit position, 0 and 1 are equally likely

* A BCH code is a multilevel, cyclic, error-correcting, variable-length digital code used to correct multiple random error patterns. BCH codes may also be used with multilevel phase-shift keying whenever the number of levels is a prime number or a power of a prime number.
BCH Code

```c
encode_bch()
{
   register int i, j;
   register int feedback;
   for (i = 0; i < length - k; i++)
      bb[i] = 0;
   for (i = k - 1; i >= 0; i--)
      {
         feedback = data[i] ^ bb[length - k - 1];
         if (feedback != 0) {
            for (j = length - k - 1; j > 0; j--)
               if (g[j] != 0)
                  bb[j] = bb[j - 1] ^ feedback;
            else
               bb[j] = bb[j - 1];
            bb[0] = g[0] && feedback;
         } else {
            for (j = length - k - 1; j > 0; j--)
               bb[j] = bb[j - 1];
            bb[0] = 0;
         }
      }
}
```

Code Profiling

- **Execute code on physical or simulated machine**
  - (Cross-)Compile down to binary
  - Run on real processor or instruction-set simulator
    - Benchmarks and input data test vectors

- **Instrument code and collect metrics at runtime**
  - Include effect of processor instruction set and architecture
    - For the given runtime platform (not necessarily the intended target)
  - Many profiling tools for data gathering and analysis
    - `gprof`, etc.
  - Various interfaces, levels of automation, and approaches to information presentation
  - A lot of work in the high performance computing community
    - Effect of instrumentation on measured results?
Instrumentation Techniques

- Program instrumentation techniques
  - Manual: Programmer inserted directives
  - Automatic: No direct user involvement
    - Sampling [gprof]
    - Binary rewriting [PIN]
    - Dynamic Instrumentation

- Hardware instrumentation techniques
  - Timers, memory system performance, processor usage, etc.
  - Available mostly through special registers or memory mapped location
    - Example: Pentium Pro MSRs (model (machine)-specific registers)
      - Counters for clock cycles, memory read/write, L1 cache misses, pipeline flushes, etc.
  - Hardware assisted trace generation

- Operating system instrumentation techniques
  - Behavior of virtual memory, file system, file cache, etc.
  - Access via APIs

- Network instrumentation techniques
  - Passive, e.g. RMON protocol packet header fields for monitoring
  - Active, e.g. ping, NWS in grid style computing.

DRM Profiling

- PC-based reference code
  - Designed to run in a Linux environment, but in this course, we are going to make it run on an embedded processor (ARM Cortex-A9)
  - Profile in both Linux and (simulated) ARM environments
    - Measure performance and identify bottlenecks

- Linux environment profiling
  - `gprof` measures where program spends its time and which functions call other functions while it was executing
    - Flat profile: total amount of time program spends executing each function
    - Call graph: how much time was spent in each function and its children

- ARM environment profiling
  - Run `gprof` in virtual platform prototype
    - Flat and call graph profiles
    - Timing accuracy of virtual platform? (QEMU timing model)
Component Simulation

- **Cycle-accurate**
  - Describe micro-architecture/RTL in C
- **Interpreted instruction-set simulation**
  - Cycle-accurate/-approximate
  - Functionality/behavior only
- **Compiled ISS**
  - Binary translation
  - Offline vs. just-in-time (self-modifying code)
  - Functional only (no/rough timing, e.g. CPI=1)
- **Source-level/host-compiled simulation**
  - C model describing functionality/behavior
  - Back-annotate with timing and other metrics

Quick EMUlator (QEMU)

- **Open-source, binary-translating ISS [Ballard’xx]**
  - Emulates a variety of architectures (x86, ARM, PowerPC)
  - Stand-alone or full-system model (peripherals to boot OS)

Source: M. Gilgor, N. Fournel, F. Pétrot, "Using Binary Translation in Event Driven Simulation for Fast and Flexible MPSoC Simulation," CODES+ISSS'09
Estimation

• Static analysis and prediction of performance
  • Need for hard guarantees in embedded space
    – Reliability, safety, etc.

➢ Worst-Case Execution Time (WCET) estimation
  ➢ Tightness of bounds?

• Analysis of modern processors is very difficult
  • Dynamic effects
    – Pipeline introduces dependencies on input sequence (instructions)
    – Cache effects
    – Branch prediction
  • Hazards lead to timing accidents & penalties
    – Structural (resource being used by another)
    – Data (dependence for data calculations)
    – Control (calculating next address – branches, interrupts)

Overall WCET Approach

➢ Compute upper bounds along the program structure
  • Programs are hierarchically structured
  • Compute the upper bound for a construct from the upper bounds of its constituents
  • Basic blocks of code at the leaves of the hierarchy

• Micro-architecture analysis
  • Abstract interpretation of code in basic blocks
  • Determines WCET for each basic block (in contexts)
  • Exclude as many timing accidents as possible

• Worst-case path determination
  • Control flow graph (CFG) of basic blocks
  • Map to an integer linear program
  • Determines upper bound and associated path
Conditional Statements

\[ A \equiv \text{if } B \text{ then } A_1 \text{ else } A_2 \]

**Constituents** of \( A \):
1. **Condition** \( B \)
2. **Statements** \( A_1 \) and \( A_2 \)

\[ wcet(A) = wcet(B) + \max(wcet(A_1), wcet(A_2)) \]

Loops

\[ A \equiv \text{for } i \leftarrow 1 \text{ to } 100 \text{ do } A_1 \]

\[ wcet(A) = wcet(i \leftarrow 1) + 100 \times (wcet(i \leq 100) + wcet(A_1)) + wcet( i \leq 100) \]
Program Path Analysis

- **Program Path Analysis**
  - Which sequence of instructions is executed in the worst-case (longest runtime)?
  - **Problem**: the number of possible program paths grows exponentially with the program length

- **Model**
  - We know the upper bounds (number of cycles) for each basic block from static analysis
  - Number of loop iterations must be bounded

- **Concept**
  - Transform structure of CFG into a set of (integer) linear equations.
  - Solution of the Integer Linear Program (ILP) yields bound on the WCET.
Control Flow Graph (CFG)

what_is_this {
1     read (a,b);
2     done = FALSE;
3     repeat {
4       if (a>b)
5          a = a-b;
6       elseif (b>a)
7          b = b-a;
8       else done = TRUE;
9     } until done;
10    write (a);
}

Control Flow Graph (CFG)

- The nodes are the basic blocks
  - Single point of entry & exit
  - Instructions in block always executed together
Calculation of the WCET

- **Definition:** A program consists of $N$ basic blocks, where each basic block $B_i$ has a worst-case execution time $c_i$ and is executed for exactly $x_i$ times. Then, the WCET is given by

$$WCET = \sum_{i=1}^{N} c_i \cdot x_i$$

- The $c_i$ values are determined using static analysis.
- How to determine $x_i$?
  - Structural constraints given by the program structure
  - Additional constraints provided by the programmer (bounds for loop counters, etc.; based on knowledge of the program context)

Example

```c
/* k >= 0 */
s = k;
WHILE (k < 10) {
  IF (ok)
    j++;
  ELSE {
    j = 0;
    ok = true;
  }
  k ++;
} 

r = j;
```
Structural Constraints

Flow equations:
- $d_1 = d_2 = x_1$
- $d_2 + d_8 = d_3 + d_9 = x_2$
- $d_3 = d_4 + d_5 = x_3$
- $d_4 = d_6 = x_4$
- $d_5 = d_7 = x_5$
- $d_6 + d_7 = d_8 = x_6$
- $d_9 = d_{10} = x_7$

Additional Constraints

Loop is executed for at most 10 times:
- $x_3 \leq 10 \cdot x_1$

B5 is executed for at most one time:
- $x_5 \leq 1 \cdot x_1$
**Integer Linear Program (ILP)**

- ILP with structural and additional constraints

\[ WCET = \max \left\{ \sum_{i=1}^{N} c_i \cdot x_i \right\} \land d_1 = 1 \land \]

\[ \sum_{j \in \text{in}(B_i)} d_j = \sum_{k \in \text{out}(B_i)} d_k = x_i, i = 1 \ldots N \land \]

- Apply standard ILP solver
  - NP-complete, i.e. exponential complexity!

**Overview**

Executable program

Control-Flow-Graph to improve WCET bounds for loops

Static Analyses

- Value Analyzer
- Cache/Pipeline Analyzer

Micro-architecture Analysis

Path Analysis

- ILP-Generator
- LP-Solver
- Evaluation

Worst-case Path Determination

WCET-Visualization

Loop-Bounds

Timing Information
Value Analysis

- **Motivation:**
  - Provide access information to data-cache/pipeline analysis
  - Detect infeasible paths
  - Derive loop bounds

- **Method:** calculate intervals at all program points, i.e. lower and upper bounds for the set of possible values occurring in the machine program (addresses, register contents, local and global variables)

- **Abstract interpretation (AI)**
  - *Semantics-based method* for static program analysis
  - Perform the program's computations using value descriptions or *abstract values* in place of the concrete values, start with a description of all possible inputs
  - Supports *correctness* proofs

---

\[
D1: [-4,4], \ A0: [0x1000, 0x1000]
\]

- Intervals are computed along the CFG edges
- At joins, intervals are "unioned"

\[
D1: [-4,4], \ A0: [0x1000, 0x1000]
\]

\[
D0: [4, 4], \ A0: [0x1000, 0x1000]
\]

\[
D0: [0, 8], \ A0: [0x1000, 0x1000]
\]

\[
D1: [-4, 4], \ A0: [0x1000, 0x1000]
\]

Which address is accessed here?

access \([0x1000, 0x1008]\)
Caches

- Caches are used, because
  - Speed gap between CPU and main memory
    ➢ Fast but expensive on-chip memory in between

- Every memory access goes through the cache
  - Block $m$ containing $a$ is in the cache (hit): request for $a$ is served in the next cycle.
  - Block $m$ is not in the cache (miss): $m$ is transferred from main memory to the cache, $m$ may replace some block in the cache, request for $a$ is served ASAP while transfer still continues.

- Several replacement strategies: LRU, PLRU, FIFO,... determine which line to replace.

Static Cache Analysis

- Abstraction
  - Sets of memory blocks in single cache lines
    ➢ From values to locations, ignoring arithmetic

  ➢ Must Analysis
    - For each program point (and calling context), find out which blocks are in the cache
    - Determines safe information about cache hits. Each predicted cache hit reduces WCET.

  ➢ May Analysis
    - For each program point (and calling context), find out which blocks may be in the cache. Complement says what is not in the cache
    - Determines safe information about cache misses. Each predicted cache miss increases BCET.
Cache with LRU: Must Analysis

**concrete**

- Access s

```
  x
  y
  t
  z
  s
  x
  y
  t
```

**abstract**

- Access s

```
{x}
()
{y, t}
()

{s}
{ }
{ }
{y, t}
```

**Age**

- "young"
- "old"

Must Analysis: Join

**Join (must)**

```
{a}
()
{c, f}
{d}

{c}
{e}
{a}
{d}
```

"intersection + maximal age"

**Interpretation:**

- Memory block `a` is definitively in the (concrete) cache => always hit
Cache with LRU: May Analysis

May Analysis: Join

Interpretation:
all blocks may be in the cache; none is definitely not in the cache.
Cache Analysis for WCET

- Information about cache contents sharpens timings

\[
\begin{align*}
\text{if } s \text{ is in must-cache:} & \\
& t_{WCET} = t_{hit} \\
& \text{otherwise} & \\
& t_{WCET} = t_{miss} \\
\text{if } s \text{ is in may-cache:} & \\
& t_{BCET} = t_{hit} \\
& \text{otherwise} & \\
& t_{BCET} = t_{miss}
\end{align*}
\]

Cache Analysis Contexts

- Cache contents depends on the context
  - Calls and loops

- First iteration loads the cache
  - Intersection loses most of the information

- Distinguish as many contexts as useful
  - 1 unrolling for caches
  - 1 unrolling for branch prediction (pipeline)
Overview

Executable program

CFG Builder

Loop Unfolding

Control-Flow-Graph

to improve WCET bounds for loops

Static Analyses

Value Analyzer

Cache/Pipeline Analyzer

Micro-architecture

Path Analysis

ILP-Generator

LP-Solver

Evaluation

Timing Information

Micro-architecture

Analysis

Worst-case Path

Determination

Motorola ColdFire 5307 Pipeline

IC1

IC2

IED

IB

EX

SST

BUS UNIT

IAG

addr(a)

hold

cancel

wait

wait

start

cancel

next

addr(a)

wait

put(a)

wait

read(A)/write(A)

fetch(a)

wait

wait

data/hold

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wait

wa
Execution of Multiply Instruction

<table>
<thead>
<tr>
<th>Fetch</th>
<th>Issue</th>
<th>Execute</th>
<th>Retire</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-Cache miss?</td>
<td>Unit occupied?</td>
<td>Multicycle?</td>
<td>Pending instructions?</td>
</tr>
</tbody>
</table>

No → 19 → 1 → 3 → 6 → 4
Yes → 30 → 6 → 6 → 41

Execution of Multiply Instruction

Cache analysis: prediction of cache hits on instruction or operand fetch or store

lwz r4, 20(r1) → Hit

Dependency analysis: analysis of data/control hazards

add r4, r5, r6
lwz r7, 10(r1)
add r8, r4, r4 → Operand ready

Resource reservation tables: analysis of resource hazards

Static Analysis of Hazards
Abstract Pipeline Execution

- \text{exec} (b : \text{basic block}, s : \text{abstract pipeline state}) t : \text{trace}
  - Interprets instruction stream of \( b \) (annotated with cache information) starting in state \( s \) producing trace \( t \)
  - \text{length}(t) gives number of cycles

- What is abstracted?
  - Abstract states may lack information, e.g. about cache contents
  - Assume local worst cases is safe (in the case of no timing anomalies)
  - Traces may be longer (but never shorter) than in reality

Context

- \textbf{Starting state for basic block? In particular, if there are several predecessor blocks?}
- \textbf{Solutions}
  - Sets of states
  - Combine by assuming that local worst case is safe
Summary of WCET Analysis Steps

- **Value analysis**

- **Cache analysis** using statically computed effective addresses and loop bounds

- **Pipeline analysis**
  - Assume cache hits where predicted,
  - Assume cache misses where predicted or not excluded.
  - Only the “worst” result states of an instruction need to be considered as input states for successor instructions!

- **Path analysis**
  - Compute final WCET estimate

Lecture 5: Outline

- **SoC design flow**
  - Iterative design convergence

- **Requirement & performance analysis**
  - Algorithm analysis
  - Component-level estimation
  - System-level estimation
**System Development**

- Get the architecture right before implementation begins
  - Best design for system performance and functionality

**Virtual Platforms**

- **Virtual prototyping**
  - Instruction-set simulation in transaction-level model (TLM)

Source: CoWare, Inc.
Architecture Analysis

Software

- ARM Core
- Instruction
- Data
- IRQ
- FIQ
- DMA Ctrl
- Slave
- Master
- Clock Gen.
- Reset Ctrl.

Bus

- AHB
- APB
- APB cfg
- Intr. Ctrl.
- Intr. Int.
- Ctrl.

Memory

- Int. ROM
- Int. RAM
- Interrupt

Hardware

- Custom Peripheral
- Input Device
- AHB
- APB

Source: CoWare, Inc.