Introduction to C-Based High Level Synthesis

Jianwen Zhu
Electrical and Computer Engineering
University of Toronto

Where is it Used? Accelerators
How many of them: a lot!

When is it used? Design Flow

- Algorithm selection optimization
- Application requirements
- HW/SW partitioning
- Behavior mapping
- Architecture exploration
- Protocol generation
- Topology synthesis

High level synthesis

- Functional model

- Architecture model

- RTL model

- RTL-to-GDSII

- Physical implementation

- Input

- Output
Why C-based

Agenda

- HLS: What is it?
  - Decision flow
  - Design representations
  - Quality metrics
- HLS: Algorithms
  - Problem formulations
  - Key algorithms
**What is HLS?**

A process of converting behavioural level design to register transfer level

**HLS Input: Behavioral level design**

- **Imperative program:**
  - Statement and expression operating on program variables
- **Characteristics**
  - Untimed
  - Captures the function of the design
  - No hardware implementation detail

```c
int A[100], B[100];
int sum, i;
sum = i = 0;
while( i < 100 ) {
    sum = sum + A[i] * B[i];
    i = i + 1;
}
```

*Dot product algorithm*
HLS Output: RTL design

- Finite state machine with datapath (FSMD)
  - Datapath performs computations (what)
  - FSM generates control flow (when)

Why HLS?

- Automate the task of RTL design
  - Similar to the transition from assembly programming to high level programming in software
- Guarantee the correctness of generated HDL
  - Correct by construction
- Value of productivity gain
  - Significant reduction of development and verification cost
  - Significant reduction of time-to-market
**HLS Design Flow**

- **Frontend and Optimizer**
  - **Frontend**: Textual program → Intermediate Representation (IR)
    - Perform lexical and syntactical analysis
  - **Optimizer**: IR → IR
    - Common sub-expression elimination
    - Dead code elimination
    - Tree height reduction
    - Strength reduction
    - ...

---

**Introduction to C-based High Level Synthesis**

Jianwen Zhu © 2009 - P. 11

---

**Introduction to C-based High Level Synthesis**

Jianwen Zhu © 2009 - P. 12
**Backend**

- HLS backend: IR $\rightarrow$ RTL
  - **Allocation**: Determine the hardware resources requirement
  - **Scheduling**: Map each operation into a **control step**, exploit the parallelism in sequential code, maintain data dependencies
  - **Binding**: Map operations to function units, map values to storage resources, exploit hardware sharing (minimize area)

---

**Code Generator**

- **RTL $\rightarrow$ HDL**
  - Translate key decisions made by backend to HDL acceptable by logic synthesis tools
  - Derive datapath from allocation and binding
  - Derive FSM from scheduling and binding
Design Representations

- Design representations
  - Capture designs with varying details
- Simplified representations for teaching
  - TinyC: simplified C language
  - TinyIR: simplified IR
  - TinyRTL: simplified RTL representation

TinyC

- Program state defined by a set of variables
- Actions defined by statements
- Simplifying assumptions
  - All statements are in a single procedure, and there are no procedure calls
  - Only support int and bool primitive types
  - Only support one-dimensional array
  - No pointers are supported
TinyC Syntax

- Defined by a set of production rules in *Backus-Naur Form* (BNF)
- Key constructs
  - Declarations: define scalars or array variables
  - Statements: assignment or control flow statements
  - Expressions: transformations of scalar, defined primitive or program variable values

TinyC in One Page

```
program:
  declaration* statement*

statement:
  variable '=' expression ';',
  'if' '(' expression ') ' statement
  ( 'else' statement )* ,
  'while' '(' expression ') ' statement ,
  'break' '; ',
  '{' declaration* statement* '}'

declaration:
  type identifier ['=' expression] ';',
  type identifier '[' expression ']'

type:
  'int', 'bool'

expression:
  '-' expression , '!' expression ,
  expression '+' expression ,
  expression '-' expression ,
  expression '*' expression ,
  expression '/' expression ,
  expression '^' expression ,
  expression '>>' expression ,
  expression '<<' expression ,
  expression '&' expression ,
  expression '|' expression ,
  expression '=' expression ,
  expression '!=' expression ,
  expression '<' expression ,
  expression '<=' expression ,
  expression '>' expression ,
  expression '>=' expression ,
  '(' expression ')' ,
  integer , identifier , 'TRUE' , 'FALSE' ,
  identifier '[' expression ']'
```
**Notations**

- A data type T corresponds to a set \( T \), in particular the integer type \( \mathbb{Z} \) corresponds to set \( \mathbb{Z} \)
- A linked list or arrays whose elements are of type T corresponds to the power set of \( T \), or the set of all subsets of \( T \), denoted as \( T^\complement \)
- A record with fields a of type A, and b of type B corresponds to a set of named tuples, denoted as \( <a: A, b: B> \)
- A graph R whose nodes are of type A corresponds to a relation \( R: A \times A \)
- A hash table or dictionary F that maps a value of type A to a value of type B corresponds to a function \( F: A \rightarrow B \)

**TinyIR**

- Why IR?
  - Decouple optimization algorithms from input languages and target architectures

**Definition:** A TinyIR is a tuple \(<O, S, V, B>\) with the following elements:
  - A set \( O = \{lds, sts, lda, sta, ba, br, cnst, +, -, *, /, <<, >>, \ldots\} \) of operation codes, which corresponds to the set of all virtual instruction types.
  - A set \( S \) of symbols, which corresponds to the scalar and array variables.
  - A set \( V: <\text{opcode}, O, \text{src1}, V, \text{src2}, V, \text{symb}, S \cup B \cup \mathbb{Z}> \) of virtual instructions, which corresponds to the expressions and control transfers in the program.
  - A set \( B: V^\complement \) of basic blocks, each containing a sequence of virtual instructions.
From TinyC to TinyIR

- Constructs in TinyC have equivalent representation in TinyIR
  - Declarations correspond to symbols
  - Statements and expressions correspond to virtual instructions
  - Virtual instructions are grouped within different basic blocks

Dot product in TinyIR

```
scalar sum;
scalar i;
array A[100];
array B[100];
B1:
  (0) cnst 0
  (1) sts (0), sum
  (2) sts (0), i
B2:
  (3) lds i

(4) lda (3), A
(5) lda (3), B
(6) * (4)(5)
(7) lds sum
(8) + (6)(7)
(9) sts (8), sum
(10) cnst 1
(11) + (3)(10)
(12) sts (11), i
(13) cnst 100
(14) < (11)(13)
(15) bt (14), B2
```
TinyRTL

- Capture key micro architectural information
  - Computational resources
  - Storage resources
- Instruction in TinyRTL is referred to as *register transfer* and is annotated with micro architecture resource use
- *Register transfer* is **cycle accurate**

---

TinyRTL Definition

**Definition:** A TinyRTL is a tuple \(<M, R, U, I, C>\) with the following elements:

- A set \(M\) of **memories** used to store scalar and array variables
- A set \(R\) of **registers** used to store scalar variables or temporary instruction results
- A set \(U\) of **functional units**, such as adders, subtractors, multipliers, shifters, etc.
- A set \(I\): \(<\text{unit}: U, \text{opcode}: O, \text{dest}: R, \text{src1}: R \cup S \cup Z \cup C, \text{src2}: R\>\) of **register transfers**, each of which uses a functional unit to transform values, which are either constants or retrieved from registers, and stores the result back to a register.
- A set \(C\) of **control steps**, each of which contains a set of register transfer
Dot Product in TinyRTL

- 5 control steps: C0-C4
- Each step contains one or more register transfers

```plaintext
register R0, R1, R2, R3;
memory M;
unit U0, U1;

C0: sts 0, R0; sts 0, R1;
C1: M.lda R2, R1, A;
C2: M.lda R3, R1, B; U0.+ R1, R1, 1;
C3: U1.* R2, R2, R3; U0.< R3, R1, 100;
C4: U0.+ R0, R0, R2; bt C1, R3;
```

FSMD

- Structured hardware descriptions
- Described in
  - Schematics
  - Verilog
  - HDL
Dot Product in FSMD: Controller

```verilog
module ctrl(
    clk, rst, status0,
    we0, we1, we2, we3,
    sel0, sel1, sel2, sel3, sel4, sel5, sel6,
    func0
);

input clk, rst;
input status0;
output we0, we1, we2, we3;
output sel0, sel1, sel2, sel3, sel4, sel5;
output [1:0] sel6;
output func0;

reg [2:0] pstate, nstate;
reg we0, we1, we2, we3;
reg sel0, sel1, sel2, sel3, sel4, sel5;
reg [1:0] sel6;
reg func0;

// present state register
always @(posedge clk or negedge rst)
if (!rst)
    pstate <= `C0;
else
    pstate <= nstate;

// next state logic
always @(pstate or status0)
case (pstate)
    `C0: nstate = `C1;
    `C1: nstate = `C2;
    `C2: nstate = `C3;
    `C3: nstate = `C4;
    `C4: if (status0)
        nstate = `C1;
    else
        nstate = `C0;
    default: nstate = `C0;
endcase
endmodule
```

// control signals
always @( pstate ) begin
we0 = 1'b0; we1 = 1'b0;
we2 = 1'b0; we3 = 1'b0;
sel0 = 1'bx; sel1 = 1'bx;
sel2 = 1'bx; sel3 = 1'bx;
sel4 = 1'bx; sel5 = 1'bx;
sel6 = 2'bxx; func0 = 1'bx;
case( pstate )
    `C0: begin
        we0 = 1'b1; we1 = 1'b1; sel0 = 1'b0;
        sel1 = 1'b0;
    end
    `C1: begin
        we2 = 1'b1; sel2 = 1'b0; sel4 = 1'b0;
    end
    `C2: begin
        we1 = 1'b1; we3 = 1'b1; sel1 = 1'b1;
        sel3 = 1'b0; sel4 = 1'b1; sel5 = 1'b0;
        sel6 = 2'b00; func0 = 1'b0;
    end
    `C3: begin
        we2 = 1'b1; we3 = 1'b1; sel2 = 1'b1;
        sel3 = 1'b1; sel5 = 1'b0; sel6 = 2'b01;
        func0 = 1'b1;
    end
    `C4: begin
        we0 = 1'b1; sel0 = 1'b1; sel5 = 1'b1;
        sel6 = 2'b10; func0 = 1'b0;
    end
endcase
end
Dot Product in FSMD: Datapath

```
Dot Product in FSMD: Datapath

```

Introduction to C-based High Level Synthesis  Jianwen Zhu © 2009 - P. 29

Dot Product in FSMD: Datapath

```
Dot Product in FSMD: Datapath

```

Introduction to C-based High Level Synthesis  Jianwen Zhu © 2009 - P. 30
Quality metrics

- When using HLS, it is important to
  - Establish required performance
  - Measure implementation performance
- Metrics need to be established to quantify both

Required Performance

- Naturally by input/output bandwidth in
  - Bits per second
  - Triangles per second
  - ...
- More often by speed in
  - Million Instructions per Second (MIPS)

\[ \text{Perf (TinyIR)} = \text{bandwidth} \times \text{work} \]

The work, measured by the number of TinyIR instructions required for the computation an algorithm applies per unit of data
**Implementation Performance**

- Use “back-of-the-envelope” method to quickly assess quality metrics directly from the RTL implementation
  
  \[ \text{Perf}(\text{TinyRTL}) = \frac{\text{work}}{\text{CycleCount}(\text{TinyRTL}) \times \text{CycleTime}(\text{TinyRTL})} \]

- **CycleCount** is number of clock cycles it takes to complete the algorithm
- **Cycle Time** is the shortest period for correct operation of the synthesized circuit

---

**Cycle Count**

- Estimate **CycleCount(TinyRTL)**
  
  - Statically examining the RTL

  **Example:** Consider the dot product algorithm in TinyRTL. It can be shown that C0 will be executed once, while C1-C4 will be executed 100 times. Therefore, \( \text{CycleCount(TinyRTL)} = 1 + 4 \times 100 = 401 \)

  Note that this is significantly less than the total number of virtual instructions, which can be calculated from TinyIR as \( 3 + 13 \times 100 = 1303 \). This type of speedup is achieved by executing multiple instructions in parallel in RTL.
**Cycle Time**

- The worst-case delay along all register-to-register paths

\[ \text{CycleTime}(\text{TinyRTL}) = \text{MAX}_{rt} \text{Delay}(rt) \]

- Process-independent delay unit

\[ FO4 = 0.36\text{ns} / \mu m * L_{\text{drawn}} \]

---

**Cycle Time**

- Dot product example, find all true register-to-register paths in FSMD model
  - Path 1: src1 → mux1 → u → muxd → dest
  - Path 2: src2 → mux2 → u → muxd → dest
  - Path 3: pstate → c1 → mux1 → u → muxd → dest
  - Path 4: pstate → c2 → mux2 → u → muxd → dest
  - Path 5: pstate → cu → u → muxd → dest
  - Path 6: pstate → cd → muxd → dest

- Note: on some cases, mux1, mux2, muxd, c1, c2, cd and cu may not be needed
Cycle Time

- Time has to be reserved for the correct functioning of the register

\[ \text{SeqOverhead} = T_{\text{setup}} + T_{cQ} + T_{\text{skew}} \]

- Now, the worst case delay of all paths in a register transfer can be calculated as:

\[
\text{Delay}(rt) = \text{SeqOverhead} + \max\left(\frac{\text{Delay}(c1) + \text{Delay}(\text{mux1})}{\text{Delay}(c2) + \text{Delay}(\text{mux2})}, \frac{\text{Delay}(c3)}{\text{Delay}(\text{mux})}\right) + \text{Delay}(a),
\]

How to estimate the delay of the multiplexers and the control logic?

- There are cases where some multiplexers or control logic are not needed
- The delay of existing multiplexers depend on the number of inputs they have
**Unit Control Signal Delay**

Assuming the delay of all control signals assume a value $T_c$, we then have:

$$\text{Delay}(cu) = \begin{cases} 0 & |\text{Opcodes}(u)| = 1 \\ T_c & |\text{Opcodes}(u)| > 1 \end{cases}$$

Where

$$\text{Opcodes}(u) = \{\text{rt.opcode} \mid \forall rt \in T[rt.u = u]\}$$

**Mux Control Signal Delay**

Similarly, we have

$$\begin{align*}
\text{Delay}(c1) &= \begin{cases} 0 & |\text{Src1}(u)| = 1 \\ T_c & |\text{Src1}(u)| > 1 \end{cases} \\
\text{Delay}(c2) &= \begin{cases} 0 & |\text{Src2}(u)| = 1 \\ T_c & |\text{Src2}(u)| > 1 \end{cases} \\
\text{Delay}(cd) &= \begin{cases} 0 & |\text{Srcd}(dest)| = 1 \\ T_c & |\text{Srcd}(dest)| > 1 \end{cases}
\end{align*}$$

where

$$\begin{align*}
\text{Src1}(u) &= \{\text{rt.src1} \mid \forall rt \in T[rt.u = u]\} \\
\text{Src2}(u) &= \{\text{rt.src2} \mid \forall rt \in T[rt.u = u]\} \\
\text{Srcd}(dest) &= \{\text{rt.dest} \mid \forall rt \in T[rt.dest = dest]\}.
\end{align*}$$
### Unit and Mux Delay

- **Process independent delay unit**
- **Pre-characterize commonly used components**

<table>
<thead>
<tr>
<th>register</th>
<th>functional unit</th>
<th>multiplexer</th>
<th>SRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>setup</td>
<td>2.0</td>
<td>adder 10.0</td>
<td>FO4</td>
</tr>
<tr>
<td>hold</td>
<td>0.0</td>
<td>comparator 6.0</td>
<td>FO4</td>
</tr>
<tr>
<td>clock skew/jitter</td>
<td>4.0</td>
<td>multiplier 35.0</td>
<td>FO4</td>
</tr>
<tr>
<td>clock to Q</td>
<td>4.0</td>
<td>inverter 1.0</td>
<td>FO4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>setup</th>
<th>2.0</th>
<th>adder 10.0</th>
<th>FO4</th>
</tr>
</thead>
<tbody>
<tr>
<td>hold</td>
<td>0.0</td>
<td>comparator 6.0</td>
<td>FO4</td>
</tr>
<tr>
<td>clock skew/jitter</td>
<td>4.0</td>
<td>multiplier 35.0</td>
<td>FO4</td>
</tr>
<tr>
<td>clock to Q</td>
<td>4.0</td>
<td>inverter 1.0</td>
<td>FO4</td>
</tr>
</tbody>
</table>

### Adding Up

**Example:** To estimate the cycle time of the dot product example, we assume $n = 5$ FO4. With the method developed previously, we can find the delay of each register transfer and determine that the cycle time of the RTL is 47.4 FO4. In 90nm technology, this is equivalent to $32.4\text{ps} \times 47.4 = 1.54\text{ns}$. In other words, the maximum speed at which the circuit can run is 649Mhz. Recall that the cycle count is 401, and the work (number of virtual instructions) is 1303, we can conclude that

$$\text{Perf} \text{(TinyRTL)} = \frac{1303}{401 \times 1.54} = 2108\text{MIPS}$$

<table>
<thead>
<tr>
<th>Register transfer</th>
<th>Seq overhead</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
<th>$d_0$</th>
</tr>
</thead>
<tbody>
<tr>
<td>slr 0, R0</td>
<td>10.0</td>
<td>0.0</td>
<td>2.4</td>
<td>5.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>slr 0, R1</td>
<td>10.0</td>
<td>0.0</td>
<td>2.4</td>
<td>5.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Msla R2, R1, A</td>
<td>10.0</td>
<td>10.5</td>
<td>2.4</td>
<td>5.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2.4</td>
</tr>
<tr>
<td>Msla R3, R1, R</td>
<td>10.0</td>
<td>10.5</td>
<td>2.4</td>
<td>5.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>2.4</td>
</tr>
<tr>
<td>Mul R1, R1, R1</td>
<td>10.0</td>
<td>10.0</td>
<td>2.4</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>M+ R2, R2, R3</td>
<td>10.0</td>
<td>10.5</td>
<td>2.4</td>
<td>5.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>4.7</td>
</tr>
<tr>
<td>Mul R3, R1, R1</td>
<td>10.0</td>
<td>10.0</td>
<td>2.4</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>2.4</td>
<td></td>
</tr>
<tr>
<td>U0+ R0, R0, R2</td>
<td>10.0</td>
<td>10.0</td>
<td>2.4</td>
<td>5.0</td>
<td>5.0</td>
<td>5.0</td>
<td>2.4</td>
<td>3.2</td>
<td>33.2</td>
</tr>
</tbody>
</table>

Introduction to C-based High Level Synthesis

Jianwen Zhu ©2009 - P. 41
High Level Synthesis Algorithms

Zooming in…

High-level synthesis algorithms

- **Focus on**
  - the backend component (i.e. core synthesis algorithms)
  - Input: IR in terms of virtual instructions
  - Output: RTL in terms of register transfers
- On top of functionally correct RTL, best **quality of result** (QoR) needs to be met as well.
## Problem Formulation

Given: \( ToyIR = \langle Q, S, Y, B \rangle \)

Find: \( ToyRTL = \langle M, R, U, I, C \rangle \)

Maximize: \( \text{Perf}(ToyRTL) \)

Minimize: \( \text{Area}(ToyRTL) \)

## Simplifying Assumptions

- User-provided functional unit allocation (i.e. the HW resources used in the implementation) → design exploration
- Partially finished storage allocation
  - Array → a single memory
  - Scalar → separate registers
  - Temp value → ONLY thing needs to be mapped
- Each virtual inst has one and only one register transfer
- Each register transfer takes a single clock
Refined Problem

Given: $\text{Try} = (O, S, V, B)$
Find:
(1) Schedule $\text{Sched} : B \mapsto (V \mapsto Z)$
(2) Register binding $B^R : V \mapsto Z$
(3) Functional unit binding $B^U : V \mapsto U$

Minimize: Objective (1) $\forall b \in B, \text{range} \text{Sched}(b)$
Objective (2) $\text{range} B^R$
Objective (3) $\sum_{s \in S} |\text{Src}(s)| + \sum_{u \in U} |\text{Src}(u)|$

Subject to: Constraint $\forall b \in B, \forall s \in S, \forall \text{Sched}(b)^{-1}(s) \leq |U|$

3 key decisions
- Scheduling: Map each inst to a control step in Sched
- Register binding: Map the value computed by each inst to a register
- Functional unit binding: Map each inst to a functional unit

Must satisfy the resource constraint
- # of inst in each ctrl step $\leq$ # of functional units

Maximize Performance
- Minimize # of control steps

Minimize Area
- Minimize # of registers and multiplexer inputs

Scheduling: Dependency Test

Why dependency test
- Schedule max # of inst in a step.
- Preserve functional correctness

Sources of data dependency
- Must
  - $A = \ldots$
  - $B = A$

- May (if $A$ & $B$ aliased to a same location)
  - (1)
    - $A = \ldots$
    - $B = \ldots$
  - (2)
    - $A = \ldots$
    - $\ldots = B$
  - (3)
    - $= A$
    - $B = \ldots$
**Dependency Test for TinyIR**

- No pointers
  - dependency test only consists of comparing the symbol names

- With pointers
  - Pointer/alias analysis has to be performed

- Test result: a **precedence graph** for each basic block

---

**Precedence Graph Example**

- TinyC(a) Code to TinyIR(b)

```c
int a, b, c, d;

// Code...

c = ...;
d = ...;

if (...) { 
  c = (a + b) * (a - b);
  d = (a + 1.2) + (a - 1.2);
}

... = v + dx;

// IR...

// Code...

// IR...
```

---
**Precedence Graph Example**

- For a basic block, for each instruction, draw its dependencies as edges in Precedence Graph.
- E.g. (28) → (26) → (24),(25)
- Instructions outside the basic block will be names “s” and “t”

![Precedence Graph Example](image)

(a) Chain of instructions
(b) Precedence Graph of B4

---

**Unconstrained Scheduling**

- Assume an unlimited # of functional units
- Problem

<table>
<thead>
<tr>
<th>Given:</th>
<th>Precedence graph ((E,s,t))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find:</td>
<td>(S : V \mapsto Z)</td>
</tr>
<tr>
<td>Minimize:</td>
<td>(S(t) - S(s))</td>
</tr>
<tr>
<td>Subject to:</td>
<td>(\forall \langle u,v \rangle \in E, S(v) - S(u) &gt; 0)</td>
</tr>
</tbody>
</table>

- Total # of steps = \(S(t) - S(s)\)
- Schedule of source must be earlier and sink must be later than any nodes in the basic block.
As Soon As Possible (ASAP) Scheduling

algorithm asapSched(\mathcal{E} : (V \times V)^\rightarrow, s : V, f : V) returns V \rightarrow Z
1. var S : V \rightarrow Z;
2. var Ready, NextReady : V[];
3. var step : Z;
4. var counter : V \rightarrow Z;
5. step = 0;
6. Ready = {s};
7. foreach (v \in V)
8.   counter(v) = \{d((u,v) \in E)\};
9. while (Ready \neq \emptyset) do
10.   NextReady = \emptyset;
11.   foreach (v \in Ready) begin
12.     S(v) = step;
13.     foreach ((v,w) \in E) begin
14.       counter(w) = counter(w) - 1;
15.       if (counter(w) == 0)
16.         NextReady = NextReady \cup \{w\};
17.     end foreach
18.   end foreach
19.   step = step + 1;
20.   Ready = NextReady;
21.   end while
22. return S;

ASAP in English

- Iterative approach
- In each iteration
  - a set of "ready" nodes (inst) are scheduled to a control step.
  - A node is ready when all its predecessors are scheduled
- Key: how to efficiently decide if a node is ready?
- Naïve approach
  - for each node being scheduled, visit each successor,
  - check all predecessors of the successor
- Better approach: keep a counter for each node representing # of predecessors
  - For each node being scheduled, visit each successor
  - Decrement the counter of the successor
  - When counter = 0, the node is ready.
  - O(|V|+|E|)
- ALAP (As Late As Possible) algorithm starts to schedule in reverse order (from sink)
**ASAP and ALAP Example**

(a) ASAP (b) ALAP (c) Mobility

Mobility shows the flexibility of scheduling for each node.

(difference between ASAP step and ALAP step.

---

**Resource-constrained Scheduling**

- Now consider with limited # of functional units:
- New constraint:
  - # of instructions scheduled at any control step must <= # of functional units
List Scheduling

Algorithm: ListSched \( E : (V \times V)^i, s : V, t : V \) returns \( V \rightarrow Z \)

1. var \( s : V \rightarrow Z \)
2. var \( NextReady : V \)
3. var \( step : Z \)
4. var \( counter : V \rightarrow Z \)
5. var \( restab : \{true, false\} \)
6. var \( step = 0 \)
7. var \( Ready = \{\} \)
8. foreach \( v \in V \)
9. \( counter(v) = \{u | (u, v) \in E\} \)
10. while \( (\text{Ready} \neq \{\}) \) do
11. \( NextReady = \{\} \)
12. foreach \( v \in U \)
13. \( restab(v) = false \)
14. while \( \exists u, y \in \text{Ready} | \text{restab}(v) \land \text{impl}(u, y) \) do
15. \( v = \text{choice}(\text{Ready}, U) \)
16. \( \text{restab}(u) = \text{true} \)
17. \( S(v) = \text{step} \)
18. \( \text{Ready} = \text{Ready} \setminus \{v\} \)
19. foreach \( (u, v) \in E \) begin
20. \( \text{counter}(u) = \text{counter}(u) - 1 \)
21. if \( \text{counter}(u) = 0 \)
22. \( \text{NextReady} = \text{NextReady} \cup \{u\} \)
23. end foreach
24. end while
25. \( \text{step} = \text{step} + 1 \)

List Scheduling in English

- Modified version of ASAP.
- Like ASAP, list of nodes ready are maintained
- Unlike ASAP
  - Unit occupancy for current control step needs to be maintained: reservation station (restab)
  - Only a subset of ready nodes can be scheduled.
- Key question:
  - How to choose the subset for better performance?
  - Classic NP-complete problem
- Solution:
  - Assign priority to ready nodes
  - Priority determined by heuristics
List Scheduling Heuristics

- Less-Flexible-First
  - assigns higher priority to nodes have smaller mobility
  - Mobility = ALAP - ASAP
- Distance to sink
- # successors

Example

(a) Uses less-flexible-first priority
(b) Random1 – did not choose 16 at step 0 (extra step)
(c) Random2 – did not choose 17 at step 2 (extra step)
Register Binding

- Virtual instructions compute values
  - Values must be stored for later use
  - Store values in registers
- Questions
  - How many registers are needed?
  - How can values be bound to registers?

Register Binding

- Objective:
  - Minimize the number of registers
  - Maximize the sharing of registers between values
- Observation:
  - Two values can only share the same register if they are not in use at the same time
**Liveness analysis**

- Objective: Determine when variables are in use
- What does it mean to be live?
- **Definition 5.5** A value (instruction) \( v \) is **live** at a control step \( s_1 \) if there exists another control step \( s_2 \) reachable from \( s_1 \), such that \( v \) is used as an operand by one of the instructions scheduled at \( s_2 \). A live set at control step \( s_1 \) is the set of all values alive at \( s_1 \).

**Example**

- Step 0: \( 15, 16, 25 \) (live values: \( 4, 6 \))
- Step 1: \( 17, 24 \) (live values: \( 4, 15, 16, 25 \))
- Step 2: \( 18 \) (live values: \( 17, 24, 25 \))
- Step 3: \( 20, 26 \) (live values: \( 18, 24, 25 \))
**Live(s), Def(s) and Use(s)**

- Liveness analysis is used to compute when variables are alive
- Uses three sets
  - Live(s): Set of live values at the beginning of step s
  - Def(s): Set of values defined at step s
  - Use(s): Set of values used at step s

**Basic Block Liveness Analysis**

- Relationship between variables
  - Live(s) = Use(s) U [Live(s+1) – Def(s)]
- Backward scan: Since each step requires information from the next step
- Idea:
  - Used variables become live
  - Defined variables become dead
**Basic Block Liveness Analysis**

\{18, 24, 25\} U [ \{20, 26\} - \{20, 26\} ]

<table>
<thead>
<tr>
<th>STEP</th>
<th>Def</th>
<th>Use</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{15, 16, 25}</td>
<td>{4, 6}</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>{17, 24}</td>
<td>{4, 15, 16}</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>{18}</td>
<td>{17}</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>{20, 26}</td>
<td>{18, 24, 25}</td>
<td>{18, 24, 25}</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>{20, 26}</td>
</tr>
</tbody>
</table>

**Basic Block Liveness Analysis**

\{17\} U [ \{18, 24, 25\} - \{18\} ]

<table>
<thead>
<tr>
<th>STEP</th>
<th>Def</th>
<th>Use</th>
<th>Live</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{15, 16, 25}</td>
<td>{4, 6}</td>
<td>{4, 6}</td>
</tr>
<tr>
<td>1</td>
<td>{17, 24}</td>
<td>{4, 15, 16}</td>
<td>{4, 15, 16, 25}</td>
</tr>
<tr>
<td>2</td>
<td>{18}</td>
<td>{17}</td>
<td>{17, 24, 25}</td>
</tr>
<tr>
<td>3</td>
<td>{20, 26}</td>
<td>{18, 24, 25}</td>
<td>{18, 24, 25}</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td>{20, 26}</td>
</tr>
</tbody>
</table>
Liveness Analysis Algorithm

- This algorithm can be extended to whole control flow graph
- Caveats:
  - CFG has backward edges
  - BB can have multiple successors
- Modifications
  - Repeatedly traverse graph until solution converges
  - Union all the live sets from predecessors before applying equation
**Liveness Analysis Algorithm**

```plaintext
algorithm live (Sched : B ↦ (V ↦ Z)) returns B ↦ (Z ↦ V[1])
1. var Live : B ↦ (Z ↦ V[1]);
2. var LiveOut : B ↦ V[1];
3. var New : V[1];
4. var changed : \{true, false\} = true;
5. while (changed) do
6.  changed = false;
7.  foreach (b ∈ B in postorder) begin
8.   New = \{ \bigcup_{op(b)} \text{Live}(s, 0) \};
9.   if (New ≠ LiveOut(b)) begin
10.  LiveOut(b) = New;
11.  changed = true;
12.  Live(b) = liveBB(Sched(b), liveOut(b));
13. end if
14. end foreach
15. end while
16. return Live;
```

---

**Interference Graph**

- Liveness of variables has been computed
- Construct a graph describing interference of two variables
  - i.e. both variables are alive at the same time
  - Nodes: represent variables
  - Edges: represent two variables alive at the same time
**Interference Graph**

![Interference Graph Diagram]

**Interference Graph Algorithm**

```plaintext
algorithm infg(Sched : B → (V → Z), Live : B → (Z → V[]) ) returns (V × V)[[]

1. var $E_{\text{infg}} : (V × V)[[]$

2. foreach ($b ∈ B$ )
   foreach ($v ∈ b$) begin
   3. $s = \text{Sched}(b, v)$;
   4. $E_{\text{infg}} = E_{\text{infg}} \cup \{(u, v), (v, u) \mid u \in \text{Live}(b, s + 1) \land u \neq v\}$;
   5. end foreach
   6. return $E_{\text{infg}}$.
```
Register binding by coloring

- Recall: We want to assign variables to registers to minimize the number of registers used.
- Corresponds to the graph coloring problem:
  - Given a graph color each node such that no edge joins nodes of the same color using the fewest number of color.

Graph Coloring

- Graph color is NP-complete.
- Need heuristics to complete problem within reasonable time.
- Overview:
  - Given a order which to visit nodes (i.e. vertex elimination order).
  - Visit each node in that order.
  - Pick a color such that no neighbor has the same one.
Example

Graph with nodes 4, 25, 15, 16.
Example

Example
Register Binding By Coloring

**Algorithm** color \((E_{ref}, (V \times V)[[]])\) returns \(V \mapsto Z\)

1. var \(C : V \mapsto Z\);
2. var \(\sigma : V \mapsto Z\);
3. var \(V' : V[]\);
4. var \(E' : E_{ref}[]\);
5. var \(v : V'\);
6. \(\sigma = \text{vertexElim} (E_{ref})\);
7. foreach \((i \in [1 \ldots |V'|])\) begin
   8. \(v = \sigma^{-1}(i)\);
   9. \(V' = V' \cup \{v\}\);
   10. \(E' = E' \cup \{(v,u) \mid u \in V'\land (u,v) \in E_{ref}\}\);
   11. \(C(v) = \min\{c \in Z \mid \forall (u,v) \in E', C(u) = c\}\);
8. endforeach
13. return \(C\);

Vertex Elimination Order

- In previous example, assumed that order was given
- For basic block
  - To determine order use **left-edge algorithm**
  - Optimal for the **interval graph**
- Generally
  - Use heuristic: less-flexible first
  - i.e. pick nodes with the most neighbors first
Vertex Elimination

- To generate vertex order
  - Visit node with fewest neighbors and push them onto a stack
  - Repeat until all edges visited
- Stack will now contain vertex order starting with the top node on the stack

Example

```
26
20
18
17
16
15
24
25
```

```
26
20
18
6
```
Example

```
4

25

15

16

```

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>24</td>
<td>18</td>
<td>17</td>
</tr>
<tr>
<td>26</td>
<td>20</td>
<td>6</td>
<td></td>
</tr>
</tbody>
</table>
**Functional Unit Binding**

- Register count and functional binding affects number of multiplexers
- Desirable have operands share registers
  - Common source: If operands for two instructions share the same register, desirable to map both instructions to the same functional unit
  - Common destination: Same as common source but applied to destination register of instruction
Impact of Register Binding on Mux

t2 = t0 + t1;
...
t5 = t3 + t4;

R0 R1 R2 R3
MUX

R0 R1
MUX

R2 R3
add

R4 R5

R0 R1
MUX

R3
add

R4 R5

Impact of Unit Binding on Mux

register R0, R1, R2, R3, R4, R5;
unit U0, U1;
C0: U0 + R4, R0, R1;
U1 + R5, R2, R3;
C1: U0 + R4, R2, R4;

U0 + U1

U1 + R5, R2, R3;

(a)

(b) (c)
Compatibility Graph

- Duality with interference graph
- Clique
  - "Complete" subgraph of a graph
- Graph coloring = Clique partitioning

Clique Partitioning Algorithm

- Given the compatibility graph
- Iteratively contract an edge
  - Merge two nodes (u and v) of the selected edge
  - Update the graph
    - Preserve only edges to nodes that are neighbors to both to u and v
Unit Binding Problem Refined

- Establish compatibility graph
  - Nodes represent instructions
  - Edges created between nodes such that
    - Opcodes are of the same class
    - Instructions are not scheduled at the same time
  - Edges are weighted
    - To help identify common source/destination
- Solve
  - weighted clique partitioning problem

Second Iteration

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>SRC 1</th>
<th>SRC 2</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>R3</td>
<td>R0</td>
<td>R2</td>
</tr>
<tr>
<td>16</td>
<td>R3</td>
<td>R0</td>
<td>R1</td>
</tr>
<tr>
<td>20</td>
<td>R2</td>
<td></td>
<td>R1</td>
</tr>
<tr>
<td>24</td>
<td>R3</td>
<td></td>
<td>R1</td>
</tr>
</tbody>
</table>
## Third Iteration

### Iteration 2

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>SRC 1</th>
<th>SRC 2</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>R3</td>
<td>R0</td>
<td>R2</td>
</tr>
<tr>
<td>16,24</td>
<td>R3</td>
<td>R0</td>
<td>R1</td>
</tr>
<tr>
<td>20</td>
<td>R2</td>
<td></td>
<td>R1</td>
</tr>
</tbody>
</table>

## End Result

### After iteration 2

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>SRC 1</th>
<th>SRC 2</th>
<th>DEST</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>R3</td>
<td>R0</td>
<td>R2</td>
</tr>
<tr>
<td>16,24,20</td>
<td>R3, R2</td>
<td>R0</td>
<td>R1</td>
</tr>
</tbody>
</table>
Random Unit Binding

Resultant Datapath

- Compare result
  - Weighted clique partitioning
  - Random clique partitioning
- On #mux inputs
  - 17
  - 20
Acknowledgement

- This powerpoint is created with the help of
  - Alex Choong
  - Zefu Dai
  - Nick Ni
  - Wai Sum Mong
- From University of Toronto

Historical Notes

- Material due to pioneering contributions of
  - McFarland, Don Thomas et al (CMU)
  - De Man et al (IMEC)
  - Parker et al (USC)
  - Gajski et al (UIUC, UC Irvine)
  - Composano et al (IBM)
  - Paulin et al (Carleton)
  - And numerous others…
- Closely related evolutions
  - Hardware/Software Codesign
  - Application-specific instruction processor
  - MPSoC / NoC
Commercial Efforts

- Early efforts
  - Synopsys Behavioral Compiler
  - Y Explorations
  - ...

- Current efforts
  - Forte
  - Synfora
  - Mentor Graphics Catapult
  - Cadence C2S
  - AutoESL
  - ...

What Went Wrong and Future Top 5

- 5: Architecture does not scale
  - FSMD limits to ILP

- 4: Application does not scale
  - Input limits in size

- 3: Poor link with logic/physical synthesis

- 2: Methodology does not scale
  - Function-centric -> Architecture-centric -> Application centric

- 1: Identity crisis: what problem to solve best and first