Compilation Techniques for VLIW Architectures

Dietmar Ebner and Florian Brandner

VU2 185.32

Exercises

Lecture 1

1. (1p) Assume masks cost $300K, and that in ASIC technology each part of a custom design costs $8.50. If you can implement the same circuit at the same speed for an FPGA that costs $70 per part, what is the break-even volume for switching over? Chart the two curves.
2. (1p) Consider a processor core that sells for $10. Assuming that the cost of an engineer/year (EY) is $200.000, how many parts do you have to sell in five years to pay for the cost of a five-person team that takes 18 months to develop the processor core? A ten-person engineering team? Assume a sales and marketing overhead of 27% of the sales cost, a build cost of $3.50 per part, ignore the time value of money.

Lecture 2

3. (3p) Select a VLIW architecture of your choice, summarize important aspects of the instruction set, and encoding. Highlight features that are explicitly hidden or exposed from/to the programmer/compiler.
4. (2p) Assume a 4-way VLIW, all operations can be encoded using 31-bits. A compiler generated a program with the following distribution of bundles:
   - 11 bundles that contain NOPs only
   - 6 bundles with only one single operation
   - 9 bundles with 2 operations
   - 44 bundles with 3 operations
   - 26 bundles with 4 operations

   Compare the uncompressed, fixed-overhead, and distributed encoding schemes, using 32-bit syllables, in terms of code size.

   Now assume that the architecture does not require vertical NOPs.

5. (2p) The Intel Itanium dedicates 5 template bits for every 128-bit bundle (of three operations). The template field specifies two properties: stops within the current bundle, and the mapping of instructions slots to execution types. Using the architecture reference manual, determine the number of possible combinations of the three operations (and their stop bits) in a bundle. What percentage of them is allowed by the 5-bit template.
6. (1p) Explain why supporting binary compatibility is problematic for VLIWs.

Lecture 3

7. (2p) Consider a program in which the dynamic execution breakdown of operations (assume unit latency) is the following:
   - 30% memory operations
   - 40% ALU operations
   - 13% branches
   - 17% floating point

   What is the maximum theoretical achievable speedup (versus a sequential machine) for a VLIW machine with one branch unit, and unlimited other resources?

   Now, assume a VLIW with two memory, four integer, two floating point, and one branch unit.
8. (2p) In a clustered VLIW, we have the choice of providing implicit or explicit copy operations, for which in the implicit case operands must include encoding bits for the full register specifier, and in the explicit case we only need a cluster bit in the encoding, at the expense of requiring extra copy operations to move values between clusters. For a 4-way VLIW architecture with 32 registers per cluster draw a figure showing the number of bits in an instruction associated with the register specifier in the two cases. Compute how many (%) intercluster copy operations can be issued before the explicit copy mechanism becomes less efficient than the implicit copy.

Lecture 4

9. (1p) Write C code that implements the multiplication of two 24-bit fractional values to produce a 48-bit fractional value, assuming 32-bit registers.
10. (5p) Assume a 4-way VLIW architecture that has a unified register file with 32-bit registers (r0-r31), and the usual arithmetic and logic operations (+,-,*,/,&,|,etc.) and a multiply-accumulate (mac), 8 predicate registers (p0-p7), and according comparison operations (==, !=, >, <, etc.). Memory can be accessed using load/store operations (ld, st). The architecture supports partial predication, and offers a `select` instruction. There are no restrictions on the instruction bundling from the architecture or the encoding. Multiply, multiply-accumulate, and load operations take at least 3 cycles, all other operations finish within 1 cycle. There is no hazard detection implemented.

Implement and optimize the following C function for the given architecture:

```
int foo(int a[], int n) {
  int i, min;
  min = UINT_MAX;
  for (i = 0; i < n; i++) {
    min = a[i] < min ? a[i] : min;
  }
  return min;
}
```

Use a simple pseudo assembly language, e.g.:

```
r1 = ld (r0);              // load
p0 = r1 != r2;             // compare
r0 = r1 + r2;              // add
r3 += r1*r2;               // mac
if (p0) goto x;            // conditional branch
goto x;                    // unconditional branch
r0 = p1 ? r2 : r3;         // select
```