“Itanium Power Programming”

Sverre Jarp
CERN openlab

S.Jarp
CERN

Agenda:

Part 1a

Introduction

Presentation Objectives

- **Offer programmers**
  - **Comprehension of the architecture**
    - **Instruction set and other features**
  - **Working Understanding of Itanium machine code**
    - **Compiler-generated code**
    - **Hand-written assembler code**

- **Inspiration for writing code**
  - **Well-targeted assembler routines**
    - **Highly optimized routines**
  - **In-line assembly code**
    - **Full control of architectural features**

Part 1b

Overview of Architecture and Conventions

Architectural Highlights

- (Some of the) Main Innovations:
  - Rich Instruction Set
  - Bundled Execution
  - Predicated Instructions
  - Large Register Files
    - Register Stack
    - Rotating Registers
  - Software Pipelined Loops
  - Control/Data Speculation
  - Cache Control Instructions
  - High-precision Floating-Point

A simple example

- **Lots of details**
  - **Many questions**

Register allocation

Application registers

Enforced Instruction Separation

```
.proc
getval:
              alloc      r3=ar.pfs,R_input,R_local,R_output,R_rotating
(p0)          movl       r2=Table                // Base table address
(p0)          and        in0=7,in0              // Choice is 0 – 7
;;
(p0)          shladd     r2=in0,3,r2            // Index table
;;
(p0)          ldfd       f8=[r2]                // Load value

(p0)          br.ret.sptk.few  rp               // return
```

Predicated execution

Branch return

User Register Overview

128 Integer Registers	16 Kernel Backup Registers
128 Floating Point Registers	8 Region Registers
64 Predicate Registers	128 Control Registers
8 Branch Registers	Instruction Pointer
128 Application Registers	NN Debug Breakpoint Registers
5 CPUID Registers	NN Perf. Mon. Data Reg’s

IA64 Common Registers

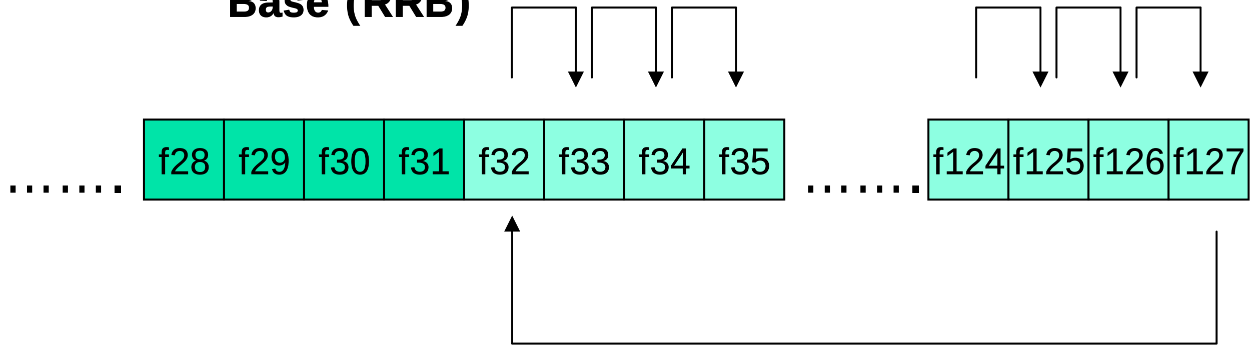
- Integer registers
  - 128 in total; Width is 64 bits + 1 bit (NaT); r0 = 0
  - Integer, Logical and Multimedia data
- Floating point registers
  - 128 in total; 82 bits wide
  - 17-bit exponent, 64-bit significand
  - f0 = 0.0; f1 = 1.0
  - Significand also used for two SIMD floats
- Predicate registers
  - 64 in total; 1 bit each (fire/do not fire)
  - p0 = 1 (default value)
- Branch registers
  - 8 in total; 64 bits wide (for address)

S.Jarp
CERN

Rotating Registers

- **Upper 75% rotate (when activated):**
  - **General registers (r32-r127)**
  - **Floating Point Registers (f32-f127)**
  - **Predicate Registers (p16-p63)**

  - **Formula:**
    - **Virtual Register = Physical Register – Register Rotation Base (RRB)**



Register Convention

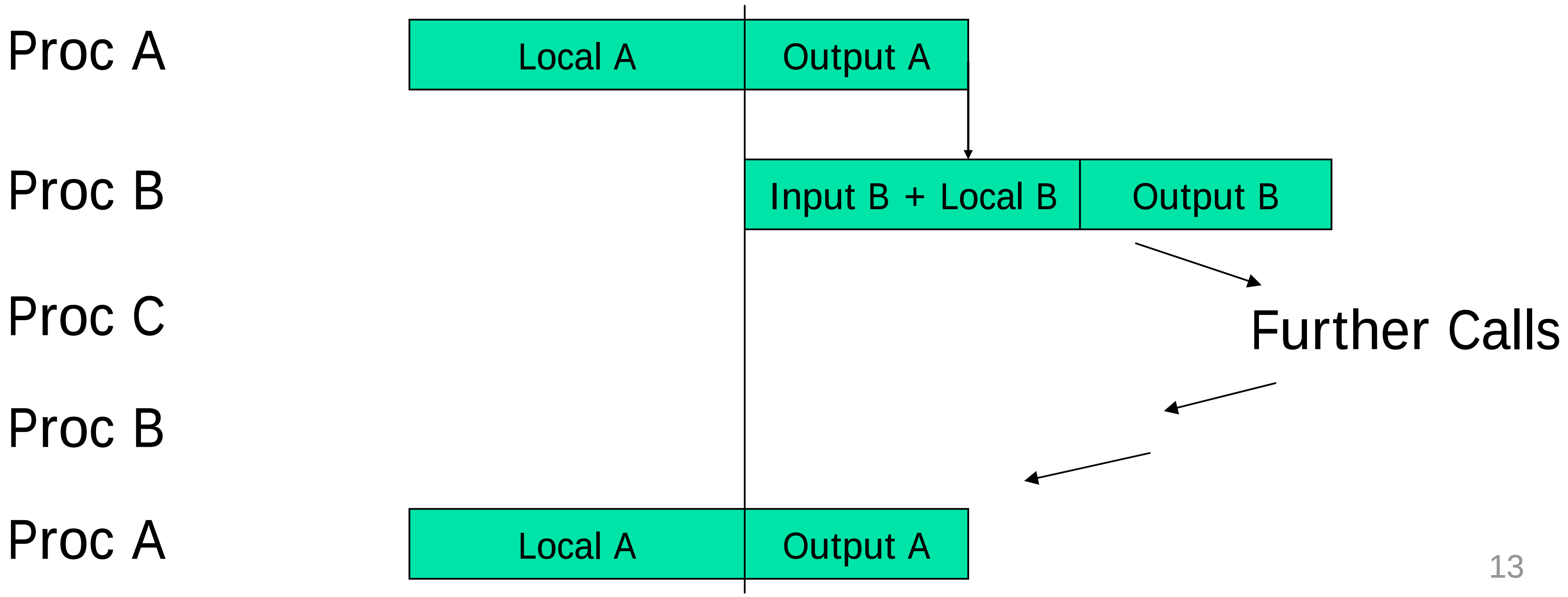
- **Run-time:**
  - **Branch Registers:**
    - **B0: Call register [rp]**
    - **B1-B5: Must be preserved**
    - **B6-B7: Scratch**
  - **General Registers:**
    - **R1: Global Data Pointer [gp]**
    - **R2-R3: scratch**
    - **R4-R7: Must be preserved**
    - **R8-R11: Procedure Return Values [ret0, ret1, ret2, ..]**
    - **R12: Stack Pointer [sp]**
    - **R13: (Reserved as) Thread Pointer**
    - **R14-R31: Scratch**
    - **R32-Rxx: Argument Registers [in0, in1, in2, ..]**

Register Convention (2)

- **Run-time convention**
  - **Floating-Point:**
    - **F2-F5: Preserved**
    - **F6-F7: Scratch**
    - **F8-F15: Argument/Return Registers**
    - **F16-F31: Must be preserved**
    - **F32-F127: Scratch**
  - **Predicates:**
    - **P1-P5: Must be preserved**
    - **P6-P15: Scratch**
    - **P16-P63: Must be preserved**
  - **Additionally:**
    - **Ar.lc: Must be preserved**

Register Stack Rules

- **The rotating integer registers serve as a stack**
  - **Each routine allocates via ”alloc” instruction:**
    - **Input + Local + Output**
    - **“R_rotate” <= “R_input + R_local” may rotate (in a multiple of 8 registers)**

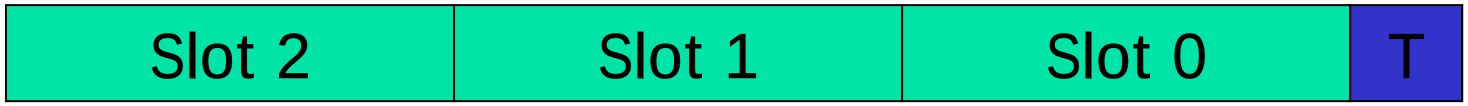


Instruction Types

- M
  - Memory/Move Operations
- I
  - Complex Integer/Multimedia Operations
- A
  - Simple Integer/Logic/Multimedia Operations
- F
  - Floating Point Operations (Normal/SIMD)
- B
  - Branch Operations
- L
  - Special instructions with 64-bit immediate

Instruction Bundle

- **Bundle as “Packaging entity”:**
  - **3 * 41 bit Instruction Slots**
  - **5 bits for Template (of Inst. types)**
    - **Typical examples: MFI or MIB**
    - **Including bit for Instruction Group Separation “S”**
  - **A bundle is 16B:**
    - **Basic unit for expressing parallelism**
    - **The unit that the Instruction Pointer points to**
    - **The unit you branch to**
    - **Actually executed may be less, equal, or more**



Instruction Group Separation (Stop bit)

- Necessary to avoid “Dependency Violations”
  - For ALL registers: Integer, FP, Predicate, Branch, App., etc.

- Two out of four possibilities (Forbidden):
  - Read-After-Write (RAW):
    - add r22=1,r21 ; add r23=1,r22 ;;
  - Write-After-Write (WAW):
    - add r22=1,r21 ; add r22=1,r23 ;;

Good assemblers will issue necessary warnings!

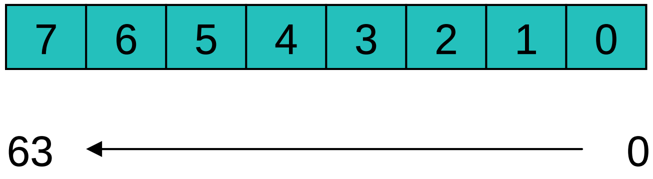
- Two out of four (OK):
  - Read-After-Read (RAR):
    - add r22=1,r21 ; add r23=1,r21 ;;
  - Write-After-Read (WAR):
    - add r23=1,r22 ; add r22=1,r21 ;;

S.Jarp
CERN

Conventions

- **Instruction syntax**
  - **(qp) ops[.comp$_1$] $r_1 = r_2, r_3$**
    - **Execution is always right-to-left**
    - **Result(s) on left-hand side of equal-sign.**
    - **Almost all instructions have a qualifying predicate**
    - **Many have further completers:**
      - **Unsigned, left, double, etc.**

- **Numbering**
  - **Also right-to left**



- **Immediates**
  - **Various sizes exist**
  - **Imm$_8$ (Signed immediate – 7 bits plus sign)**

At execution time, sign bit is extended all the way to bit 63

Part 2a

Standard Instruction Set

The Total Instruction Set

- **Many Instruction Categories:**
  - **Logical operations (e.g. and)**
  - **Arithmetic operations (e.g. add)**
  - **Compare operations**
  - **Shift operations**
  - **Branches, including loop control**
  - **Memory and cache operations**
  - **Move operations**

  - **Multimedia operations (e.g. padd)**

  - **Floating Point operations (e.g. fma)**
  - **SIMD Floating Point operations (e.g. fpma)**

See documentation for complete reference set

S.Jarp
CERN

Arithmetic Operations

- **Instruction format:**
  - (qp) ops_1 $r_1 = r_2, r_3[,1]$
  - (qp) ops_2 $r_1 = imm_x, r_3$
  - (qp) ops_3 $r_1 = r_2, count_2, r_3$

X86 Inc/Dec
replaced with
(qp) ops $r_1 = r_2$,r0,1

Z = Y – imm
becomes
(qp) Add r_1 =-imm, r_3

- **Valid Operations:**
  - ops_1: add, sub
  - ops_2: sub, adds/addl (imm_{14}, imm_{22})
  - ops_3: shladd

Loading
an immediate value
(qp) Add r_1 =imm, r0

  - NB: Integer multiply is an FLP operation

Compare Operations

- **Instruction format:**
  - (qp) cmp.crel.ctype $p_1, p_2 = r_2, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = imm_8, r_3$
  - (qp) cmp.crel.ctype $p_1, p_2 = r0, r_3$

Parallel inequality form →

- **Valid Relationships:**
  - eq, ne, lt, le, gt, ge, ltu, leu gtu, geu,
- **Types:**
  - *none*, unc, and, or, or.andcm, orcm, andcm, and.orcm

S.Jarp
CERN

Load Operations

- **Standard instructions:**
  - **(qp) ld*sz*.ldtype.ldhint $r_1=[r_3], r_2$**
  - **(qp) ld*sz*. ldtype.ldhint $r_1=[r_3], imm_9$**
  - **(qp) ldf*fsz*.fldtype.ldhint $f_1=[r_3], r_2$**
  - **(qp) ldf*fsz*.fldtype.ldhint $f_1=[r_3], imm_9$**

Always post-modify

- **Valid Sizes:**
  - **sz: 1 / 2 / 4 / 8 [bytes]**
  - **fsz: s(ingle) / d(double) / e(extended) / 8(as integer)**

Sign-bit is NOT extended for 1 / 2 / 4 bytes

In the case of integer multiply (for instance)

- **Types:**
  - **s / a / sa / c.nc / c.clr / c.clr.acq / acq / bias**
    - **Advanced options (not discussed here!)**

Also “fill” variants

More complex usage (see Manuals)

Branch Operations

- Several different types:
  - Conditional or Call branches
    - Relative offset (IP-relative) or Indirect (via branch registers)
    - Triggered by predication
  - Return branches
    - Indirect + Qualifying Predicate (QP)
  - Loop controlling branches:
    - Simple Counted Loops (br.cloop)
      - IP-relative with AR.LC
    - Software-pipelined Counted Loop (br.ctop)
      - IP-relative with AR.LC and AR.EC
    - Software-pipelined While Loops (br.wtop)
      - IP-relative with QP and AR.EC

Simple Counted Loop

- **Works as 'expected'**
  - **ar.lc counts down the loop (automatically)**
    - **No need to use a general register**

```
            mov         ar.lc=5  ;; // NB: 6 iterations
loop:       { work }
             .......
            { much more work }
            br.cloop.sptk.few     loop ;;
```

- **Software-pipelined loops are more advanced**
  - **Uses Epilogue Count (as well as Loop Count)**
  - **… and Rotating Registers**

We will deal with such loops later

One use of predication

- Avoid cost of branching
  - Which can be high due to misprediction

```
If (b > 0) b++;
else b--;
```

  - Both b++ and b– are done in the same cycle:

```
        cmp.gt.unc    p6,p7=r2,0     ;;
(p6)     add      r2=1,r2
(p7)     add      r2=-1,r2                  ;;
```

Part 2b

Our first “real” example

Expressing a loop

- Use array search example, “find”, to demonstrate how to get started
  - Based on background information on registers and conventions
  - First with a basic counted loop and later more advanced versions

```
int find(int key, int n, int* vect)
{
   int i;
   for (i=0; i<n; ++i)
   {
    if (key == vect[i]) return i;  // Found
   }
   return -1; // Not found
}
```

The loop itself

- **Simple counted loop**
  - **Only five instructions**
  - **Use input registers directly**
  - **Main latency is the load latency**
  - **NB: In the same cycle we can have Compare + Related branch**

```
cntloop:
        ld4      r31=[in2],4
        add      ret0=1,ret0      // tracking of index
;;
        cmp4.eq.unc   p6,p0=s_temp,in0
(p6)    br.cond,dpnt.few     found
        br.cloop.dptk.few    cntloop
;;
```

Total “search” program – V.1

- **Initial version:**
  - **Classical “counted loop”**
  - **Minimal:**
    - **Register usage**
    - **Assembler directives**
    - **Entry/Exit code**
  - **Main latency in loop**
    - **From “ld4”**

```
#define s_pfssave  r9
#define s_lcsave    r10
#define s_temp      r31
#define Name find
.text
.global Name
.type   Name,@function
.proc   Name
Name:
          alloc   s_pfssave=ar.pfs,3,0,0,0
          mov     s_lcsave=ar.lc
          cmp.le.unc p6,p0=in1,r0
(p6)      br.cond.dpnt.few  notfound   ;;
          add      in1=-1,in1    ;;     // loop count - 1
          mov      ret0=-1              // index count
          mov      ar.lc=in1  ;;        // loop count
cntloop:
          ld4      s_temp=[in2],4
          add       ret0=1,ret0  ;;    // track index
          cmp4.eq.unc    p6,p0=s_temp,in0
(p6)       br.cond.dpnt.few     found
          br.cloop.dptk.few     cntloop  ;;
//
notfound:    mov     ret0=-1   ;;      //Not found
found:   mov    ar.lc=s_lcsave
           br.ret.sptk.many    rp
.endp
```

Part 3a

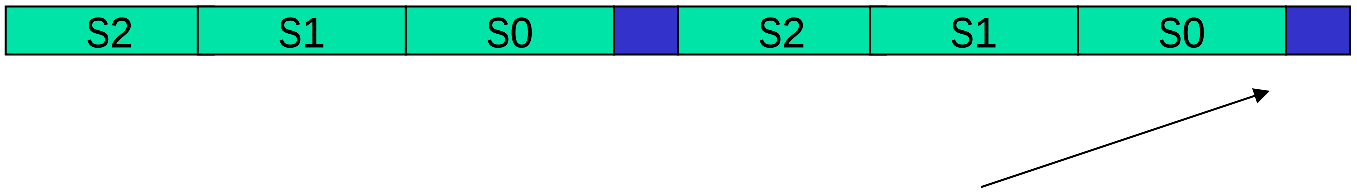
Secrets of speed

Key Performance Enablers

- Exploit
  - Architectural support
    - Memory optimization:
      - Prefetching, Load pair instructions, Branch-Predict, etc.
    - Modulo Scheduling support
      - Predication (“loop control”)
      - Register Rotation (Large Register Files)
    - Predication (“if-conversion”)
    - Vectorisation
      - Integer/FLP SIMD
  - Micro-architecture
    - Consistent, Wide execution:
      - Number of parallel bundles; Execution units; Latencies
    - Memory specifications:
      - Cache sizes, Bandwidth

Itanium Execution Width

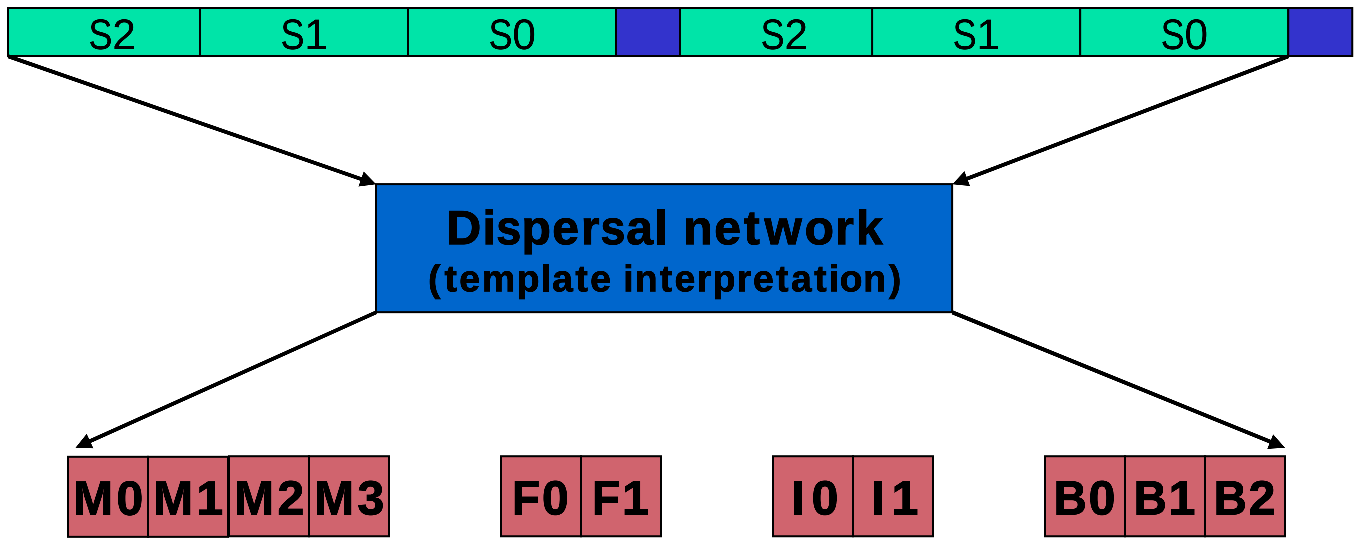
- A given IA-64 implementation could be N wide
  - All Itanium processors are implemented as a "two-banger"
    - 6 parallel instructions
      - More parallelism than IA-32
    - But,
      - If nothing useful is put into the syllables, they get filled as NOPs



This template should be even (i.e. without stop bit)

Instruction Delivery

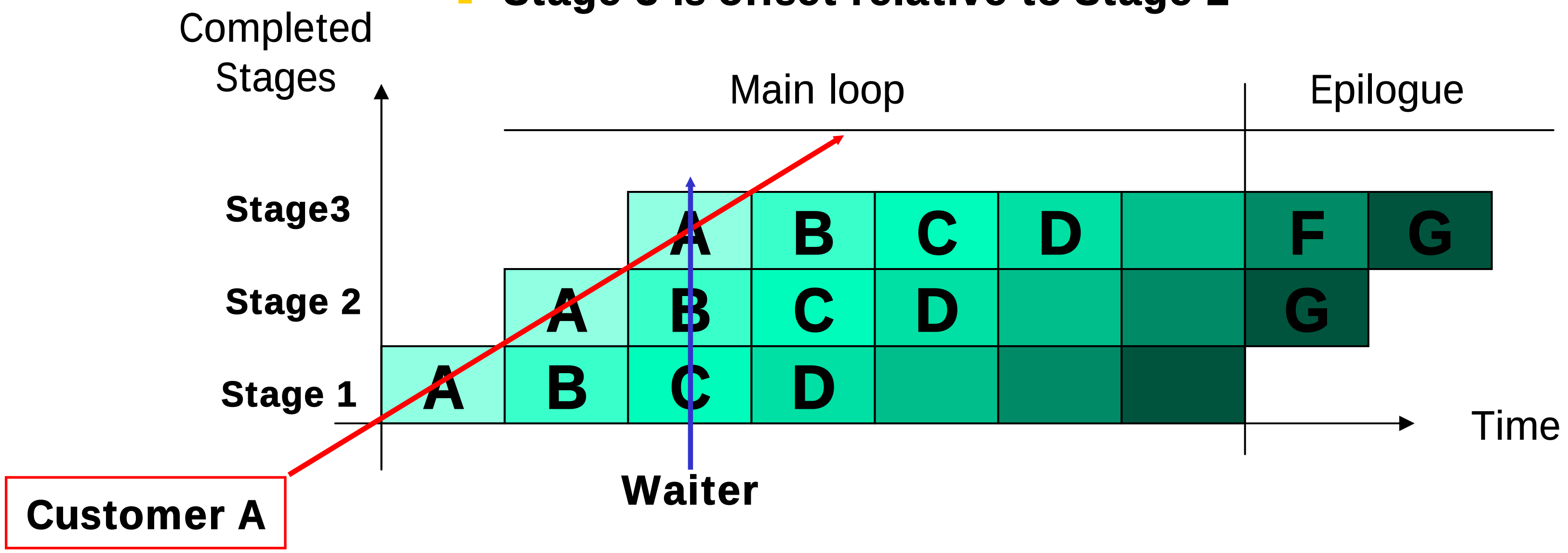
- **Must match**
  - **instructions to issue ports**
    - **w/corresponding execution units attached**



11 available ports in total

Software-pipelined loops

- **Graphical representation**
  - **N loop traversals desired, but with skewed execution:**
    - **Stage 2 is offset relative to Stage 1**
    - **Stage 3 is offset relative to Stage 2**



Analogy: Think of a restaurant where each customer (Red arrow) wants to: 1) order food, 2) eat the meal, 3) pay the bill.

The waiter (Blue arrow) is working "flat out" by

1) taking the order from C, 2) serving the meal to B, 3) getting paid by A.

Modulo Loops

- **How is it programmed ?**
  - **By using:**
    - **Rotating registers (Programmable renaming)**
      - **Let register contents live longer**
    - **Predication**
      - **Each stage uses a distinct predicate register starting from p16**
        - **Stage 1 controlled by p16**
        - **Stage 2 by p17**
        - **Etc.**
    - **Architected loop control using BR.CTOP**
      - **Clock down LC & then EC**
      - **Set p16 = 1 when LC > 0**
      - **Set P16 = 0 otherwise**

Part 3b

- **Back to our “find” example:**
  - **We are now ready to try to produce a software pipelined loop**

```
int find(int key, int n, int* vect)
{
   int i;
   for (i=0; i<n; ++i)
   {
    if (key == vect[i]) return i;  // Found
   }
   return -1; // Not found
}
```

Step 3: Pipelined loop

- **One cycle loop:**
  - **Possible when 6 (or fewer) instructions**
  - **All latencies are hidden**
  - **No dependency violations (no stops)**
    - **Due to rotating registers**

```
                  mov    s_key=in0
                  mov    s_pvect=in2  // must be moved
;;
modloop:
(p16)        ld4    r32=[s_pvect],4
(p17)        add    ret0=1,ret0          // easy tracking of index
(p17)        cmp4.eq.unc p6,p0=r33,s_key
(p6)         br.cond.dpnt.few  found
             br.ctop.sptk.few modloop
;;
```

S.Jarp

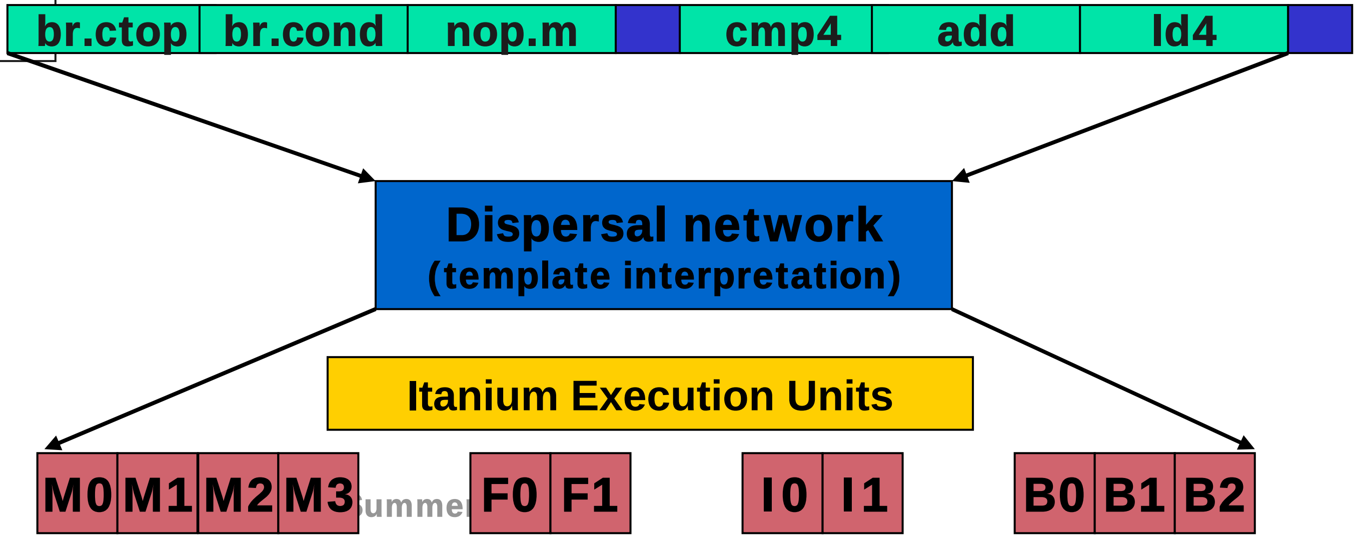
Advanced Topics:

- Tight coding:
  - Manual bundling
  - Verification against available execution units

```
modloop:
{ .mii
pc[0]     ld4      array[0]=[s_pvect],4
pc[LL]    add      ret0=1,ret0    // easy tracking
pc[LL]    cmp4.eq.unc  qc[0],p0=array[LL],s_key
}
{ .mbb
          nop.m 0
qc[CL]    br.cond.dpnt.few  found
          br.ctop.sptk.few modloop   ;; }
```

Next question:

How can we double the speed of this routine ?



br.ctop
br.cond
nop.m
cmp4
add
ld4
Dispersal network
(template interpretation)
Itanium Execution Units
M0
M1
M2
M3
F0
F1
I0
I1
B0
B1
B2