SPU Code Development and Optimization
SPE

- 128 128-bit registers
- even pipeline – arithmetic
- odd pipeline – load, store, shuffle
- 256 KB Local Store (load / store – 6 cycles latency)
- Memory Flow Controller – DMA to / from system memory
DUAL ISSUE

- Hazard detection and stall
- Dual-issue rules
  - Independent instructions
  - Even pipeline – even work address
  - Odd pipeline – odd word address

<table>
<thead>
<tr>
<th>Pipe 0 Instructions</th>
<th>Stall (clocks)</th>
<th>Latency (clocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single precision floating-point operations</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Integer multiplies, convert between float/integer, interpolate</td>
<td>0</td>
<td>7</td>
</tr>
<tr>
<td>Immediate loads, logical ops, integer add/subtract, signed extend, count leading</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>zeros, select bits, carry/borrow generate</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Double precision floating-point operations</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Element rotates and shifts</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Byte operations – count ones, abs difference, average, sum</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pipe 1 Instructions</th>
<th>Stall (clocks)</th>
<th>Latency (clocks)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shuffle bytes, quadword rotates and shifts</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Gather, mask, generate insertion control</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Estimate</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Loads/stores, branch hints</td>
<td>0</td>
<td>6</td>
</tr>
<tr>
<td>Branches</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>Channel operations, move to/from SPRs</td>
<td>0</td>
<td>6</td>
</tr>
</tbody>
</table>
SPE (User) Registers

- **0** – dedicated – link register
- **1** – dedicated – stack pointer
- **2** – volatile – environment pointer (for languages that need one)
- **3-79** – volatile – function arguments, return values, and general usage
- **80-127** – non-volatile – used for local variables
  - must be preserved across function calls
Short Vector SIMD

➢ 2 doubles – including FMA – not fully pipelined – 4 FLOP / 7 cycles
➢ 4 floats – including FMA – fully pipelined – 8 FLOP / cycle
➢ 4 integers
➢ 8 short integers
➢ 16 chars
Preferred Slot

- addresses
- lengths of shifts and rotations
- .....
AOS vs. SOA

union {
    struct _coordinate {
        float x, y, z, w;
    } coordinate;
    vector float vertex;
} array[];

struct {
    float x[];
    float y[];
    float z[];
    float w[];
} vertices;
Shuffle

- splats
- lookup table
- transposition
- endian reversal
- transform from AOS to SOA
vector unsigned char shufflehi = VEC_LITERAL(vector unsigned char,
  0x00, 0x01, 0x02, 0x03, 0x10, 0x11, 0x12, 0x13,
  0x04, 0x05, 0x06, 0x07, 0x14, 0x15, 0x16, 0x17);

vector unsigned char shufflelo = VEC_LITERAL(vector unsigned char,
  0x08, 0x09, 0x0A, 0x0B, 0x18, 0x19, 0x1A, 0x1B,
  0x0C, 0x0D, 0x0E, 0x0F, 0x1C, 0x1D, 0x1E, 0x1F);

aibj = spu_shuffle(abcd, ijkl, shufflehi);
cdkl = spu_shuffle(abcd, ijkl, shufflelo);
emfn = spu_shuffle(efgh, mnop, shufflehi);
gohp = spu_shuffle(efgh, mnop, shufflelo);

abcd = spu_shuffle(aibj, emfn, shufflehi);
efgh = spu_shuffle(aibj, emfn, shufflelo);
ijkl = spu_shuffle(cdkl, gohp, shufflehi);
mnop = spu_shuffle(cdkl, gohp, shufflelo);

Shuffle – 4 × 4 Transposition
Shifts and Rotations

- only shift / rotate left
- register or immediate

- quadword – odd pipeline
  - by bits – max 7 positions
  - by bytes – max 32 positions – shift by more than 15 bits zeros the quadword

- element – even pipeline
  - word (32 bits)
    - by bits – 64 positions – shift by more than 31 bits zeros the word
  - halfword (16 bits)
    - by bits – 32 positions – shift by more than 15 bits zeros the halfword
Branch Hint

- fetches branch target into the hint buffer
- no penalty for correctly predicted branches
- compiler inserts hints where beneficial

HINT branch, target

BRANCH if true

<table>
<thead>
<tr>
<th>Program Feature</th>
<th>Static Prediction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unconditional branch</td>
<td>always</td>
</tr>
<tr>
<td>Conditional branch</td>
<td>never</td>
</tr>
<tr>
<td>Loop-iteration branch</td>
<td>always</td>
</tr>
<tr>
<td>Loop-termination branch</td>
<td>never</td>
</tr>
<tr>
<td>Call and return</td>
<td>always</td>
</tr>
<tr>
<td>Pointer comparison</td>
<td>false</td>
</tr>
<tr>
<td>Less than zero</td>
<td>false</td>
</tr>
</tbody>
</table>
Hinting Branches

- only one HBR instruction can be active at a time; issuing another HBR cancels the current one
- an HBR instruction can be moved outside of a loop and will be effective on each loop iteration as long as another HBR or sync instruction is not executed
- the HBR instruction must be placed within 64 instructions of the branch instruction
- the HBR instruction only affects performance
- both branch and branch hint are in the odd pipeline
Hinting Branches

```c
if(__builtin_expect((a>b), 0))
    c += a;
else
    d += 1;
```
Avoiding Branches

unsigned int a, b, c;
if (a > b) d += a;
else d += 1;

clgt cc, a, b
brz cc, else
then:
a br done
else:
ai done
done:

clgt cc, a, b
a d_plus_a, d, a
ai d_plus_1, d, 1
selb d, d_plus_1, d_plus_a, cc
## SIMD Data Types

<table>
<thead>
<tr>
<th>Vector Data Type</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>vector unsigned char</td>
<td>Sixteen 8-bit unsigned chars</td>
</tr>
<tr>
<td>vector signed char</td>
<td>Sixteen 8-bit signed chars</td>
</tr>
<tr>
<td>vector unsigned short</td>
<td>Eight 16-bit unsigned halfwords</td>
</tr>
<tr>
<td>vector signed short</td>
<td>Eight 16-bit signed halfwords</td>
</tr>
<tr>
<td>vector unsigned int</td>
<td>Four 32-bit unsigned words</td>
</tr>
<tr>
<td>vector signed int</td>
<td>Four 32-bit signed words</td>
</tr>
<tr>
<td>vector unsigned long long</td>
<td>Two 64-bit unsigned doublewords</td>
</tr>
<tr>
<td>vector signed long long</td>
<td>Two 64-bit signed doublewords</td>
</tr>
<tr>
<td>vector float</td>
<td>Four 32-bit single-precision floats</td>
</tr>
<tr>
<td>vector double</td>
<td>Two 64-bit double precision floats</td>
</tr>
<tr>
<td>qword</td>
<td>quadword (16-byte)</td>
</tr>
</tbody>
</table>
**SPU Intrinsics**

- **specific intrinsics – si_____**
  - intrinsics that have a one-to-one mapping with a single assembly-language instruction

- **generic intrinsics – spu_____**
  - intrinsics that map to one or more assembly-language instructions as a function of the type of input parameters

- **composite intrinsics – spu_____**
  - convenience intrinsics constructed from a sequence of specific or generic intrinsics

<table>
<thead>
<tr>
<th>Intrinsic</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spu_mfcdma32(is, ea, size, tagid, cmd)</td>
<td>Initiate DMA to or from 32-bit effective address</td>
</tr>
<tr>
<td>spu_mfcdma64(is, eahi, ealow, size, tagid, cmd)</td>
<td>Initiate DMA to or from 64-bit effective address</td>
</tr>
<tr>
<td>spu_mfcstat(type)</td>
<td>Read MFC tag status</td>
</tr>
</tbody>
</table>
si_ Intrinsics

- one-to-one mapping with assembly
- SPU assembly instruction prefixed with si_
- also available as generic intrinsics except for:
  - generate controls for sub-quadword insertions
  - constant formation
  - NOPs
  - load and store
  - control
- accompanied by casting si_ intrinsics – no effect on the data

---

**e.g.:**

<table>
<thead>
<tr>
<th>Constant Formation Intrinsics</th>
<th>Immediate load word</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = si_il(imm)</td>
<td>Immediate load word</td>
</tr>
<tr>
<td>d = si_iia(imm)</td>
<td>Immediate load address</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>No Operation Intrinsics</th>
<th>No operation (load)</th>
</tr>
</thead>
<tbody>
<tr>
<td>si_Inop()</td>
<td>No operation (load)</td>
</tr>
<tr>
<td>si_nop()</td>
<td>No operation (execute)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Memory Load and Store Intrinsics</th>
<th>Load quadword (a form)</th>
</tr>
</thead>
<tbody>
<tr>
<td>d = si_lqa(imm)</td>
<td>Load quadword (a form)</td>
</tr>
<tr>
<td>d = si_lqd(a, imm)</td>
<td>Load quadword (d form)</td>
</tr>
<tr>
<td>Intrinsic</td>
<td>Description</td>
</tr>
<tr>
<td>---------------------------</td>
<td>--------------------------------------------------</td>
</tr>
<tr>
<td><strong>Constant Formation Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_splats(a)</td>
<td>Replicate scalar a into all elements of vector d</td>
</tr>
<tr>
<td><strong>Conversion Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_convtf(a, scale)</td>
<td>Convert integer vector to float vector</td>
</tr>
<tr>
<td><strong>Arithmetic Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_add(a, b)</td>
<td>Vector add</td>
</tr>
<tr>
<td><strong>Byte Operation Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_avg(a, b)</td>
<td>Vector average</td>
</tr>
<tr>
<td><strong>Compare, Branch, and Halt Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_cmpeq(a, b)</td>
<td>Vector compare equal</td>
</tr>
<tr>
<td><strong>Bit and Mask Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_shuffle(a, b, pattern)</td>
<td>Shuffle bytes of a vector</td>
</tr>
<tr>
<td><strong>Logical Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_and(a, b)</td>
<td>Vector bit-wise AND</td>
</tr>
<tr>
<td><strong>Rotate Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_rli(a, count)</td>
<td>Element-wise bit rotate left</td>
</tr>
<tr>
<td><strong>Shift Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_shl(a, count)</td>
<td>Element-wise bit shift left</td>
</tr>
<tr>
<td><strong>Control Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>(void) spu_idisable()</td>
<td>Disable interrupts</td>
</tr>
<tr>
<td><strong>Scalar Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_extract(a, element)</td>
<td>Extract vector element from vector</td>
</tr>
<tr>
<td><strong>Channel Control Intrinsics</strong></td>
<td></td>
</tr>
<tr>
<td>d = spu_readch(channel)</td>
<td>Read word channel</td>
</tr>
</tbody>
</table>
SPU Optimization

```c
void spu_code_tile(float *A, float *B, float *C) {
    int m, n;

    for (n = 0; n < 32; n++)
        for (m = 0; m < 32; m++)
            C[m] -= A[n*32+m] * B[n];
}
```

standard C

- good news: will compile and run
- bad news: no performance whatsoever
SPU Optimization

0D  01
1D  0123
0D  12
1D  1
0D  --45
1D  456789
0   56
0   67
0D  78
1D  789012
0D  89
1D  890123
0D  90
1D  901234
1    0123
1   --3456
1    4567
1    5678
0   67
0   --901234
1    -----5678
1   ---901234
0D   01
1D   0123
0D  0 12
0   23
0D   34
1D   --5678
1   6789
ori  $14,$4,0
shlqbyi  $15,$3,0
il  $13,0
lnop
.L2:  ori  $8,$15,0
hbra  .L11,.L3
il  $6,0
il  $9,32
.L3:  a  $3,$6,$5
lqx  $12,$6,$5
a  $2,$13,$14
lqd  $4,0($8)
ai  $9,$9,-1
lqx  $11,$13,$14
cwx  $10,$6,$5
rotqby  $3,$12,$3
rotqby  $4,$4,$8
rotqby  $2,$11,$2
ai  $8,$8,4
fnms  $3,$4,$2,$3
shufb  $7,$3,$12,$10
stqx  $7,$6,$5
ai  $6,$6,4
ceqi  $2,$13,128
brnz  $9,.L3
ai  $13,$13,4
ai  $15,$15,128
ceqi  $2,$13,128
brz  $2,.L2
bi  $lr

0.24 GFLOPS
<1% peak!!!
void spu_code_tile(float *A, float *B, float *C)
{
    vector float *Ap = (vector float*)A;
    vector float *Cp = (vector float*)C;
    int m, n;

    for (n = 0; n < 32; n++)
    {
        vector float b_splat = spu_splats(B[n]);

        for (m = 0; m < 8; m++)
        {
            Cp[m] = spu_nmsub(Ap[n*8+m], b_splat, Cp[m]);
        }
    }
}
### SPU Optimization

<table>
<thead>
<tr>
<th>OP Code</th>
<th>Instruction</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0D 01</td>
<td>ori</td>
<td>$10, $3, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 0123</td>
<td>shlqbyi</td>
<td>$12, $4, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D 12</td>
<td>il</td>
<td>$11, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 1234</td>
<td>fsmbi</td>
<td>$9, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D 23</td>
<td>ila</td>
<td>$13, $66051</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 2</td>
<td>lnop</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D -45</td>
<td></td>
<td>.L2: a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 456789</td>
<td>hbra</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D 56</td>
<td></td>
<td>.L12, .L3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 567890</td>
<td>il</td>
<td>$6, $0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 67</td>
<td></td>
<td>lqx $3, $11, $12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d -1234</td>
<td>il</td>
<td>$7, $8</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 5678</td>
<td></td>
<td>rotqby $2, $3, $2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d 2</td>
<td></td>
<td>nop $127</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d -45</td>
<td></td>
<td>shufb $8, $2, $2, $13</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 456789</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D 567890</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 67</td>
<td></td>
<td>.L3: a</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d 678901</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 78</td>
<td></td>
<td>lqx $4, $6, $5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d -890123</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d -45</td>
<td></td>
<td>ai $7, $7, $1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d 456789</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0D 67</td>
<td></td>
<td>lqx $3, $9, $2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 1234</td>
<td></td>
<td>fnms $3, $3, $8, $4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 12</td>
<td></td>
<td>stqx $3, $6, $5</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1D 1234</td>
<td></td>
<td>ai $6, $6, $16</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>.L12: brnz $7, .L3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td>ai $11, $11, $4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0d 34</td>
<td></td>
<td>ai $9, $9, $128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d 45</td>
<td></td>
<td>ceqi $2, $11, $128</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1d -6789</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>brz $2, .L2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>bi $lr</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**1.08 GFLOPS**

4.5x faster

~4% peak
```c
void spu_code_tile(float *A, float *B, float *C)
{
    vector float *Ap = (vector float*)A;
    vector float *Cp = (vector float*)C;
    vector float c0, c1, c2, c3, c4, c5, c6, c7, c8;
    int n;

    c0 = Cp[ 0];   c4 = Cp[ 4];
    c1 = Cp[ 1];   c5 = Cp[ 5];
    c2 = Cp[ 2];   c6 = Cp[ 6];
    c3 = Cp[ 3];   c7 = Cp[ 7];

    for (n = 0; n < 32; n++)
    {
        vector float b_splat = spu_splats(B[n]);
        c0 = spu_nmsub(Ap[n*8+ 0], b_splat, c0);
        c1 = spu_nmsub(Ap[n*8+ 1], b_splat, c1);
        c2 = spu_nmsub(Ap[n*8+ 2], b_splat, c2);
        c3 = spu_nmsub(Ap[n*8+ 3], b_splat, c3);
        c4 = spu_nmsub(Ap[n*8+ 4], b_splat, c4);
        c5 = spu_nmsub(Ap[n*8+ 5], b_splat, c5);
        c6 = spu_nmsub(Ap[n*8+ 6], b_splat, c6);
        c7 = spu_nmsub(Ap[n*8+ 7], b_splat, c7);
    }

    Cp[ 0] = c0;   Cp[ 4] = c4;
    Cp[ 1] = c1;   Cp[ 5] = c5;
}
```

**SPU Optimization**

**SIMD'zation + unrolling**
- unroll loops
- use independent instructions
- reuse registers between iterations
- eliminate innermost loop completely
SPU Optimization

7.64 GFLOPS
32 × faster
~30% peak
SPU Optimization

further optimizations
➢ loop pipelining / register double buffering
   ➢ compute even iteration
   / load store odd iteration
➢ loop tiling for improved locality
➢ loop linearization
   ➢ replace nested loops with a single loop
     + pointer arithmetic

Basic Loop
for $i = 0$ to $n$
- Load $i$
- Compute $i$
- Store $i$

Software-Pipelined Loop
Pipeline 0 (even)
- Load 0
- Compute 0
- Load $i + 1$
- Store $i - 1$
- Compute $n$
- Store $n$
SPU Optimization

c0 = Cp[0];    a0 = Ap[0];    c1 = Cp[1];    a1 = Ap[1];
c2 = Cp[2];    a2 = Ap[2];    c3 = Cp[3];    a3 = Ap[3];
c4 = Cp[4];    a4 = Ap[4];    c5 = Cp[5];    a5 = Ap[5];
c6 = Cp[6];    a6 = Ap[6];    c7 = Cp[7];    a7 = Ap[7];
b_splat = spu_splats(B[0]);

for (n = 0; n < 32; n+=2)
{
    d_splat = spu_splats(B[n+1]);
    c0 = spu_nmsub(a0, b_splat, c0);   e0 = Ap[n*8+8];
    c1 = spu_nmsub(a1, b_splat, c1);   e1 = Ap[n*8+9];
    c2 = spu_nmsub(a2, b_splat, c2);   e2 = Ap[n*8+10];
    c3 = spu_nmsub(a3, b_splat, c3);   e3 = Ap[n*8+11];
    c4 = spu_nmsub(a4, b_splat, c4);   e4 = Ap[n*8+12];
    c5 = spu_nmsub(a5, b_splat, c5);   e5 = Ap[n*8+13];
    c6 = spu_nmsub(a6, b_splat, c6);   e6 = Ap[n*8+14];
    c7 = spu_nmsub(a7, b_splat, c7);   e7 = Ap[n*8+15];

    b_splat = spu_splats(B[n+2]);
    c0 = spu_nmsub(e0, d_splat, c0);   a0 = Ap[n*8+16];
    c1 = spu_nmsub(e1, d_splat, c1);   a1 = Ap[n*8+17];
    c2 = spu_nmsub(e2, d_splat, c2);   a2 = Ap[n*8+18];
    c3 = spu_nmsub(e3, d_splat, c3);   a3 = Ap[n*8+19];
    c4 = spu_nmsub(e4, d_splat, c4);   a4 = Ap[n*8+20];
    c5 = spu_nmsub(e5, d_splat, c5);   a5 = Ap[n*8+21];
    c6 = spu_nmsub(e6, d_splat, c6);   a6 = Ap[n*8+22];
    c7 = spu_nmsub(e7, d_splat, c7);   a7 = Ap[n*8+23];
}

Cp[0] = c0;    Cp[1] = c1;    Cp[2] = c2;    Cp[3] = c3;
### SPU Optimization

**12.34 GFLOPS**

- ~51 x faster
- ~48% peak
SPU Decrementer

- 32 bit counter
- counting down
- blade: 14,318 MHz – 223 cycles min – 5 minutes max
- PS3: 79,800 MHz – 40 cycles min – 54 seconds max

<table>
<thead>
<tr>
<th>SPU Decrementer</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>spu_read_decrementer()</td>
<td>Read the current value of the decrementer</td>
</tr>
<tr>
<td>spu_write_decrementer(count)</td>
<td>Load a value into the decrementer</td>
</tr>
</tbody>
</table>
SPU Optimization

asmVis
- split odd and even pipeline
- color coding of dependencies
- instruction rearrangement
  preserving dependencies
SPU Optimization

asm development
➢ independent odd and even pipeline editing
➢ automated color coding
➢ zoom-in zoom-out