High-performance Regular Expression Scanning on the Cell/B.E. Processor

Daniele Paolo Scarpazza,
Multicore Solutions Dept.
dpscarpazza@us.ibm.com

Gregory F. Russell,
Scalable Systems Dept.
gfr@us.ibm.com

IBM T. J. Watson Research Center
Summary

- Matching Regular Expressions (RE): a very common, expensive workload

- Traditional tools do not generate efficient RE scanners; no multiple threads, no SIMD

- We present a technique:
  - to perform fast, parallel RE scanning with flex-compatible semantics, using all the levels of parallelism, including SIMD;
  - with an optimized implementation for the Cell/B.E.
    - performance: 8.33 – 14.30 Gbps/chip
    - speedup: 57 – 81 times faster than flex (Cell/B.E.),
    - limit: machine states ≤ 1000 – 6000
What is the value of this?

- RE scanning (tokenization) is the first stage of any search engine indexer (~20%), XML engine (~30%), programming language parser;

- it is an inherently sequential, control-dominated task, slow, difficult to optimize (especially on the Cell)

- the kernel we present is **SIMDized**, branchless, w/ single-table automaton, align/pad, manual SW pipelined and it is the fastest software tokenizer known to us

- future architectures will have wider SIMD units: e.g., Intel AVX, 256-bit SIMD
  - Intel Larrabee, 512-bit SIMD + vector loads/stores
our technique benefits from increasing SIMD width!
What is regular expression matching?

Input text

... traded yesterday in London at $1,295 an ounce, its highest in almost nine months. Palladium has rallied 35 percent this year to $254 an ounce. The metals are also used in jewelry and ...

Matches (output)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>traded</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>yesterday</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>in</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>London</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>at</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>$1,295</td>
</tr>
</tbody>
</table>

Rule set

1. \[a-zA-Z_][a-zA-Z0-9_]*
2. ([0-9]{5}-){5}[0-9]{5}
3. \$[0-9]{1,3}(,[0-9]{3})*\.[0-9]+)?
How are scanners generated traditionally?

Rule set
1. [a-zA-Z_][a-zA-Z0-9_]*
2. ([0-9]{5}-){5}[0-9]{5}
3. \$[0-9]{1,3}(,[0-9]{3})*(\.[0-9]+)?

flex / Jflex / re2c / ...

tables

stock code

compiler

Input text
... traded yesterday in London at $1,295 an ounce, its highest in almost nine months. Palladium has rallied 35 percent this year to $254 an ounce. The metals are also used in jewelry and ...

Matches (output)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>traded</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>yesterday</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>in</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>London</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
<td>at</td>
</tr>
<tr>
<td>6</td>
<td>3</td>
<td>$1,295</td>
</tr>
</tbody>
</table>

... ... ...
We are here

**Rule set**

1. \[a-zA-Z_][a-zA-Z0-9_]*\n2. ([0-9]{5}-){5}[0-9]{5}\n3. \$[0-9]{1,3}(,[0-9]{3})*\(\.[0-9]+\)?

**Input text**

... traded yesterday in London at $1,295 an ounce, its highest in almost nine months. Palladium has rallied 35 percent this year to $254 an ounce. The metals are also used in jewelry and ...

**Matches (output)**

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>traded</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td>yesterday</td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>in</td>
</tr>
<tr>
<td>3</td>
<td></td>
<td>London</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td>at</td>
</tr>
<tr>
<td>5</td>
<td></td>
<td>$1,295</td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

... ...
## Assumptions, Strengths and Limitations

### Using Flex
- **stable, mature tool**
- **wide range of RE operators**
- **familiar to many developers**
- **longest-of-the-leftmost semantics, with priority**
- **backup transitions**

### Storing the Automaton in Core-Local Memory Only
- **low latency**
- **extreme performance**
- **state-transition table limited in size (<< 256k)**

### Matches Well the Requirements of Tokenization
longest-of-the-leftmost semantics

\( i[a-z]^*n \)

internationalization

(1,2) in
(1,6) intern
(1,11) internation
(1,20) internationalization
(9,11) ion
(9,20) ionalization
(14,20) ization
(17,20) ion
flex on the Cell/B.E.

• flex is inefficient on the SPEs
  – two tables
    next state, associated semantic action
    at least two table lookups per state transition
  – irregular control flow
  – see example >>
10

Simple example

yy_start = 1; /* start state */
unsigned char * end = text_buf_end;
yy_cp = text_buf;
while ( yy_cp < end ) /* loops until end-of-file is reached */
{ 
    yy_bp = yy_cp;
    yy_current_state = (yy_start);
    yy_match:
        while ( (yy_current_state = yy_nxt[yy_current_state][UI(*(yy_cp))] ) > 0 )
            if ( yy_accept[yy_current_state] ) {
                yy_last_accepting_state = yy_current_state;
                yy_last_accepting_cpos = yy_cp;
            }
            yy_cp = tolower(*yy_cp);
            ++yy_cp;
        yy_current_state = -yy_current_state;

    yy_find_action:
    yy_act = yy_accept[yy_current_state];

do_action: /* This label is used only to access EOF actions. */
    switch ( yy_act )
    {
    case 0:
        /* must back up */
        /* undo the effects of YY_DO_BEFORE_ACTION */
        yy_cp = (yy_last_accepting_cpos) + 1;
        yy_current_state = (yy_last_accepting_state);
        goto yy_find_action;
    case 1: //STOP WORD -- JUST IGNORE IT
        /*printf("stop word: "); print_token( data, yy_bp, yy_cp ); printf("\n");
        break;
    case 2:
        add_token(yy_bp, yy_cp);
        break;
    case 3: // genitive -- remove the last two characters "'s"
        add_token(yy_bp, yy_cp-2);
        break;
    case 4:
        add_token(yy_bp, yy_cp);
        break;
    case 5: {
        const unsigned new_len = kill_dots(yy_bp, yy_cp);
        add_token(yy_bp, yy_bp + new_len );
    }
        break;
    ...
    case YY_END_OF_BUFFER:
        return;
    default:
        fprintf(stderr, "fatal flex scanner internal error--no action %1 found\n", yy_act );
        /* end of action switch */
        break;
    } /* end of scanning one token */
yy_start = 1; /* start state */
unsigned char * end = text_buf_end;
yy_cp = text_buf;
while ( yy_cp < end ) /* loops until end-of-file is reached */
{
    yy_bp = yy_cp;
    yy_current_state = (yy_start);
    yy_match:
        while ( (yy_current_state = yy_nxt[yy_current_state][UI(*yy_cp)]) > 0 )
            if ( yy_accept[yy_current_state] )
                yy_last_accepting_state = yy_current_state;
                yy_last_accepting_cpos = yy_cp;
            *yy_cp = tolower(*yy_cp);
            ++yy_cp;
        yy_current_state = -yy_current_state;

    yy_find_action:
        yy_act = yy_accept[yy_current_state];

    do_action: /* This label is used only to access EOF actions. */
        switch ( yy_act )
        {
            case 0:
                /* must back up */
                /* undo the effects of YY_DO_BEFORE_ACTION */
                yy_cp = (yy_last_accepting_cpos) + 1;
                yy_current_state = (yy_last_accepting_state);
                goto yy_find_action;

            case 1: //STOP WORD -- JUST IGNORE IT
                //printf("stop word: "); print_token( data, yy_bp, yy_cp ); printf("\n");
                break;

            case 2:
                add_token(yy_bp, yy_cp);
                break;

            case 3: /* genitive -- remove the last two characters \'s\*/
                add_token(yy_bp, yy_cp-2);
                break;

            case 4:
                add_token(yy_bp, yy_cp );
                break;

            case 5:
                /* const unsigned new_len = kill_dots(yy_bp, yy_cp); */
                add_token(yy_bp, yy_bp + new_len );
                break;

            default:
                fprintf(stderr, "fatal flex scanner internal error--no action %d found\n", yy_act );
                /* end of action switch */
            /* end of scanning one token */
Dual cycle: 5.6%
Branch miss: 24.6%
Dependency stalls: 37.2%
CPI = 2.12

Best-case performance:
  212.4 cycles/char
  15.1 MB/s / SPE
  0.96 Gbps / chip

Typical performance:
  330.5 cycles/char
  9.7 MB/s / SPE
  0.62 Gbps / chip
Summary of the techniques we apply

- Transforming the state transition graph
  - final to initial shortcuts,
  - resolve ambiguous backups,

- Transforming the state transition table
  - use a single-table encoding
  - resolve most pointer arithmetics at compile time

- Turn all control flow into data flow
  - replace branches with predicated instructions (sel_bits)
  - reduce the cost of back-ups and restores
  - write output speculatively

- Apply parallelism: 8x SPEs, 4-way SIMD units

- Low level optimizations:
  - data alignment/padding
  - manual software pipelining of loads/stores
  - code doubling to reduce dependency stalls
Summary of the techniques we apply

- Transforming the state transition graph
  - final to initial shortcuts,
  - resolve ambiguous backups,

- Transforming the state transition table
  - use a single-table encoding
  - resolve most pointer arithmetics at compile time

- Turn all control flow into data flow
  - replace branches with predicated instructions (sel_bits)
  - reduce the cost of back-ups and restores
  - write output speculatively

- Apply parallelism: 8x SPEs, 4x SIMD units

- Low level optimizations:
  - data alignment/padding
  - manual software pipelining of loads/stores
  - code doubling to reduce dependency stalls
... faster pointer arithmetics

```
next_state = table [current_state] [input]
```

```
table_base + current_state * row_size + input * cell_size
```

ADD    SHL    ADD    SHL
... faster pointer arithmetics

How to:
1. custom link script
2. STT content relocation at startup
3. input SHL1, highly SIMDizable, 16-way
Summary of the techniques we apply

• Transforming the state transition graph
  – final to initial shortcuts,
  – resolve ambiguous backups,

• Transforming the state transition table
  – use a single-table encoding
  – resolve pointer arithmetics at compile time

• Turn all control flow into data flow
  – replace branches with predicated instructions (sel_bits)
  – reduce the cost of back-ups and restores
  – write output speculatively

• Apply parallelism: 8x SPEs, 4-way SIMD units

• Low level optimizations:
  – data alignment/padding
  – software pipelining of loads/stores
  – code doubling to reduce dependency stalls
SIMD parallelization

- data flow of 4 automata, performing 4 independent transitions

Do not really study now!
Just note that:

- all black circles are SIMD instructions (1 instruction, 4 engines)
- no branches
- benefits from wider SIMD units
Simple example: Partial branch removal

```c
while ( 1 )
{
    const unsigned char in_chr = *yy_cp;
    const unsigned short next_code = yy_next[yy_current_state][in_chr];
    const unsigned short next_state = next_code >> 8;
    const unsigned is_final = next_code & BIT_FINAL;
    const unsigned is_save = next_code & BIT_SAVE;
    const unsigned is_restore = next_code & BIT_RESTORE;
    const unsigned is_token = next_code & BIT_TOKEN;
    const unsigned is_acronym = next_code & BIT_ACRONYM;

    yy_cp += (is_final ? 0 : 1);
    yy_last_accepting_cpos = is_save ? yy_cp : yy_last_accepting_cpos;
    if (UNLIKELY(yy_cp == text_buf_end)) goto get_out;
    yy_current_state = next_state;
    YY_cp = is_restore ? yy_last_accepting_cpos : yy_cp;

    if ( is_token ) {
        if ( UNLIKELY(is_acronym) ) {
            const unsigned new_len = kill_dots(yy_bp, yy_cp);
            add_token(yy_bp, yy_bp + new_len);
        } else add_token(yy_bp, yy_cp);
    }

    yy_bp = is_final ? yy_cp : yy_bp;
}
```c
while ( 1 )
{
    const unsigned char in_chr = *yy_cp;
    const unsigned short next_code = yy_next[yy_current_state][in_chr];
    const unsigned short next_state = next_code >> 8;

    const unsigned is_final = next_code & BIT_FINAL;
    const unsigned is_save = next_code & BIT_SAVE;
    const unsigned is_restore = next_code & BIT_RESTORE;
    const unsigned is_token = next_code & BIT_TOKEN;
    const unsigned is_acronym = next_code & BIT_ACRONYM;

    yy_cp += (is_final ? 0 : 1);
    yy_last_accepting_cpos = is_save ? yy_cp : yy_last_accepting_cpos;
    if (unlikely(yy_cp == text Buf end)) goto get_out;
    yy_current_state = next_state;
    yy_cp -= is_restore ? yy_last_accepting_cpos : yy_cp;

    if (is_token)
    {
        if (unlikely(is_acronym))
        {
            const unsigned new_len = kill dots(yy_bp, yy_cp);
            add_token(yy_bp, yy_bp + new_len);
        }
        else
        {
            add_token(yy_bp, yy_cp);
        }
    }

    yy_bp = is_final ? yy_cp : yy_bp;
}
```

### Performance Metrics

**Dual Cycle:**
- Branch miss: 5.6%
- Dependency stalls: 37.2%
- CPI = 2.12
- Best-case performance: 212.4 cycles/char, 15.1 MB/s / SPE, 0.96 Gbps / chip
- Typical performance: 330.5 cycles/char, 9.7 MB/s / SPE, 0.62 Gbps / chip

**Dual Cycle:**
- Branch miss: 8.1%
- Dependency stalls: 34.1%
- CPI = 1.34
- Best-case performance: 66.2 cycles/char, 48.4 MB/s / SPE, 3.09 Gbps / chip
- Typical performance: 104.6 cycles/char, 30.6 MB/s / SPE, 1.96 Gbps / chip
## CPI: a fundamental efficiency indicator

<table>
<thead>
<tr>
<th>CPI</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.50</td>
<td>100%</td>
</tr>
<tr>
<td>0.65</td>
<td>77%</td>
</tr>
<tr>
<td>0.75</td>
<td>67%</td>
</tr>
<tr>
<td>1.00</td>
<td>50%</td>
</tr>
<tr>
<td>1.25</td>
<td>40%</td>
</tr>
<tr>
<td>1.5</td>
<td>33%</td>
</tr>
<tr>
<td>2.0</td>
<td>25%</td>
</tr>
<tr>
<td>5.0</td>
<td>10%</td>
</tr>
</tbody>
</table>
Speculative output update

```
class
const unsigned char in_chr = *l_cp;
const unsigned short * p_next = ((unsigned short)(l_state)) + in_chr;
unsigned short next = *p_next;
const unsigned short state_freeze = l_state;

l_cp += 1-(is_final / BIT_FINAL);
l_acp = is_save ? l_cp : l_acp;
next = ( l_cp >= l_buf_end ) ? state_freeze : next;
l_cp = ( l_cp >= l_buf_end ) ? l_buf_end : l_cp;
l_state = KILL_FLAGS(l_state);
l_cp = is_restore ? l_acp : l_cp;

const vector unsigned qw_ptt_entry = { (unsigned)l_bp, (unsigned)l_cp, 0, 0};
* (vector unsigned * __restrict__) l_ptt_cur = qw_ptt_entry;
* (vector unsigned * __restrict__) l_acro_cur = qw_ptt_entry;

l_ptt_cur += ( is_token / BIT_TOKEN );
l_acro_cur += ( is_acronym / BIT_ACRONYM);
l_bp = is_final ? l_cp : l_bp;
```
Dual cycle: 5.6%  Dual cycle: 8.1%  Dual cycle: 12.1%
Branch miss: 24.6%  Branch miss: 2.1%  Branch miss: 0.0%
Dependency stalls: 37.2%  Dependency stalls: 34.1%  Dependency stalls: 37.8%
CPI = 2.12  CPI = 1.34  CPI = 1.21

Best-case performance:
212.4 cycles/char 66.2 cycles/char 49.4 cycles/char
15.1 MB/s / SPE 48.4 MB/s / SPE 64.8 MB/s / SPE
0.96 Gbps / chip 3.09 Gbps / chip 4.15 Gbps / chip

Typical performance:
330.5 cycles/char 104.6 cycles/char 78.1 cycles/char
9.7 MB/s / SPE 30.6 MB/s / SPE 41.0 MB/s / SPE
0.62 Gbps / chip 1.96 Gbps / chip 2.62 Gbps / chip

const unsigned char inchr = *l_cp;  
const unsigned short *p_next = ((unsigned short)(l_state)) + inchr;  
unsigned short next = *p_next;  
const unsigned short state_freeze = l_state;

l_cp += 1-(is_final / BIT_FINAL );  
l_acp = is_save ? l_cp : l_acp;  
next = ( l_cp >= l_buf_end ) ? state_freeze : next;  
l_cp = ( l_cp >= l_buf_end ) ? l_buf_end : l_cp;  
l_state = KILL_FLAGS(l_state);  
l_cp = is_restore ? l_acp : l_cp;

const vector unsigned qw_ptt_entry = { (unsigned)1_bp, (unsigned)1_cp, 0, 0};  
* (vector unsigned *__restrict__) l_ptt_cur = qw_ptt_entry;  
* (vector unsigned *__restrict__) l_acro_cur = qw_ptt_entry;

l_ptt_cur += ( is_token / BIT_TOKEN );  
l_acro_cur += ( is_acronym / BIT_ACRONYM);  
l_bp = is_final ? l_cp : l_bp;
const vector unsigned qw_in_char = {in_chr_0_p1, in_chr_1_p1, in_chr_2_p1, in_chr_3_p1};
const variant_t v_nextp = { .qw = spu_add(v_state.qw, qw_in_char) };  
vector unsigned qw_next = { (unsigned) * v_nextp.usp[0], (unsigned) * v_nextp.usp[1], 
const vector unsigned qw_state_freeze = v_state.qw;

// 1_cp
++= 1-(is_final / BIT_FINAL );
const vector unsigned is_final = spu_and( qw_next, qw_BIT_FINAL);
const vector unsigned is_token_incr16 = spu_and(qw_next, qw_BIT_TOKEN);
v_cp.qw = spu_add ( v_cp.qw, spu_sub(qw.ones, is_final ));

// 1_acp
             = is_save ? 1_cp : 1_acp;
v_lacp.qw = spu_sel ( v_lacp.qw, v_cp.qw, /*pattern=*/ spu_cmpgt(spu_and(qw_next, qw_BIT_SAVE), qw_zeros) );
const vector unsigned is_final_mask32 = spu_cmpgt(is_final, qw_zeros);
const vector unsigned qw_not_at_end_mask32 = spu_cmpgt(v_buf_end.qw, v_cp.qw);
v_state.qw = spu_sel( qw_state_freeze, qw_next, qw_not_at_end_mask32 );
v_cp.qw = spu_sel( v_buf_end.qw, v_cp.qw, qw_not_at_end_mask32 );
v_state.qw = spu_and( v_state.qw, qw_KILL_FLAGS );
const variant_t v_is_acronym = { .qw = spu_and(qw_next, qw_BIT_ACRONYM) };

// 1_cp
             = is_restore ? 1_acp : 1_cp;
const vector unsigned is_restore_mask32 = spu_cmpgt( spu_and(qw_next, qw_BIT_RESTORE), qw_zeros);
v_cp.qw = spu_sel( v_cp.qw, v_lacp.qw, is_restore_mask32 );
in_chr_0_p1 = * (v_cp.ucp[0]); /* Anticipating input load */
in_chr_1_p1 = * (v_cp.ucp[1]);
in_chr_2_p1 = * (v_cp.ucp[2]);
in_chr_3_p1 = * (v_cp.ucp[3]);

const vector unsigned qw_ptt_entry_0 = { v_bp.u[0], v_cp.u[0], v_is_acronym.u[0], 0};
const vector unsigned qw_ptt_entry_1 = { v_bp.u[1], v_cp.u[1], v_is_acronym.u[1], 0};
const vector unsigned qw_ptt_entry_2 = { v_bp.u[2], v_cp.u[2], v_is_acronym.u[2], 0};
const vector unsigned qw_ptt_entry_3 = { v_bp.u[3], v_cp.u[3], v_is_acronym.u[3], 0};

//1_bp = is_final ? 1_cp : 1_bp;
v_bp.qw = spu_sel( v_bp.qw, v_cp.qw, is_final_mask32 );

si_stqd( (qword) qw_ptt_entry_0, (qword) v_ptt_cur.qw,
si_stqd( (qword) qw_ptt_entry_1, (qword) spu_rlqwb( v_ptt_cur.qw, 4 ), 0 );
si_stqd( (qword) qw_ptt_entry_2, (qword) spu_rlqwb( v_ptt_cur.qw, 8 ), 0 );
si_stqd( (qword) qw_ptt_entry_3, (qword) spu_rlqwb( v_ptt_cur.qw, 12 ), 0 );

//1_ptt_cur += ( is_token / BIT_TOKEN );
v_ptt_cur.qw = spu_add( v_ptt_cur.qw, is_token_incr16 );

if ( spu_extract(spu_orx(qw_not_at_end_mask32),0) == 0 ) goto get_out;
const vector unsigned qw_in char = [in char 0, in char 1, in char 2, in char 3];
const variant_t v_nextp = [v_nexts.add2, v_nexts.char, v_nexts.char];
vector unsigned qw_next = [unsigned] * v_nextp.usp[0], [unsigned] * v_nextp.usp[1], [unsigned] * v_nextp.usp[2], [unsigned] * v_nextp.usp[3];
const vector unsigned qw_state_freeze = [v_state qw]

// l cp
++ l = (is final) 
const vector unsigned is_final = [spu and (qu qw, qu BIT FINAL)];
const vector unsigned is_clean_incr = [spu_and (qu qw, qu BIT_FINAL)];
v qw = spu_add (v cp, spu sub (qu one, is_final));

// l cp
is_saver = ? l cp : l cp;
v lcp qw = spu sel (v lcp qw, v cp qw, //patternc = spu cmpgt(spu and (qu qw, qu BIT _RAISE), qu zeroes));
const vector unsigned is_new_mask = [spu cmpgt(is final, qu zeroes)];
const vector unsigned qw_not at_end mask = [spu cmpgt(v buf_end qw, v cp qw)];
v qw = spu sel (v state qw, v state freez, qu next, qu not at end mask);
const vector unsigned v cp qw = spu sel (v buf_end qw, v cp qw, qu not at end mask);
v qw = spu and (v state qw, v fill flag);
const variant_t v_is_ACRONYM = [v qw, spu and (qu next, qu BIT _ACRONYM)];

// l cp
is restore mask = ? l cp : l cp;

in char 0 = [v cp usp[0]]; // Anticipating input load
in char 1 = [v cp usp[1]]; // Anticipating input load
in char 2 = [v cp usp[2]]; // Anticipating input load
in char 3 = [v cp usp[3]]; // Anticipating input load

const vector unsigned qw_ptr_entry 0 = [v bp ul, v cp ul, v cp ul, v is ACRONYM ul];
const vector unsigned qw_ptr_entry 1 = [v bp ul, v cp ul, v cp ul, v is ACRONYM ul];
const vector unsigned qw_ptr_entry 2 = [v bp ul, v cp ul, v cp ul, v is ACRONYM ul];
const vector unsigned qw_ptr_entry 3 = [v bp ul, v cp ul, v cp ul, v is ACRONYM ul];

// l cp = is final ? l cp : l cp;
v lcp qw = spu sel (v bp cp qw, v cp qw, lcp qw, is final mask);

if (spu_extract(spu ors (qu not at end mask)), 0) = 0 goto get out;

Dual cycle: 5.6% Dual cycle: 8.1% Dual cycle: 12.1% Dual cycle: 32.4%
Branch miss: 24.6% Branch miss: 2.1% Branch miss: 0.0% Branch miss: 0.0%
Dependency stalls: 37.2% Dependency stalls: 34.1% Dependency stalls: 37.8% Dependency stalls: 7.4%
CPI = 2.12 CPI = 1.34 CPI = 1.21 CPI = 0.75

Best-case performance:
212.4 cycles/char 66.2 cycles/char 49.4 cycles/char 18.6 cycles/char
15.1 MB/s / SPE 48.4 MB/s / SPE 64.8 MB/s / SPE 172.2 MB/s / SPE
0.96 Gbps / chip 3.09 Gbps / chip 4.15 Gbps / chip 11.02 Gbps / chip

Typical performance:
330.5 cycles/char 104.6 cycles/char 78.1 cycles/char 29.4 cycles/char
9.7 MB/s / SPE 30.6 MB/s / SPE 41.0 MB/s / SPE 108.9 MB/s / SPE
0.62 Gbps / chip 1.96 Gbps / chip 2.62 Gbps / chip 6.97 Gbps / chip
Code doubling

```c
vector unsigned qw_not_at_end_mask32;
const vector unsigned qw03_in_char = in_chr_0_pl, in_chr_1_pl, in_chr_2_pl, in_chr_3_pl;
const vector unsigned qw47_in_char = in_chr_4_pl, in_chr_5_pl, in_chr_6_pl, in_chr_7_pl;
const variant t v83_nextp = { .vu = spu.add(v83_state.vu, qw83_in_char) };
const variant t v47_nextp = { .vu = spu.add(v47_state.vu, qw47_in_char) };
vector unsigned qw03_next = { (unsigned) * v83_nextp.usp[0], (unsigned) * v83_nextp.usp[1], (unsigned) * v83_nextp.usp[2], (unsigned) * v83_nextp.usp[3] };
vector unsigned qw47_next = { (unsigned) * v47_nextp.usp[0], (unsigned) * v47_nextp.usp[1], (unsigned) * v47_nextp.usp[2], (unsigned) * v47_nextp.usp[3] };

const vector unsigned v83_is_final = spu.and( qw03_next, qw_BIT_FINAL);
const vector unsigned v47_is_final = spu.and( qw47_next, qw_BIT_FINAL);
const vector unsigned v83_is_token_incr16 = spu.and( qw03_next, qw_BIT_TOKEN);
const vector unsigned v47_is_token_incr16 = spu.and( qw47_next, qw_BIT_TOKEN);

v83 cp.vu = spu.add( v83 cp.vu, spu.sub(qw ones, v83 is_final ));
v47 cp.vu = spu.add( v47 cp.vu, spu.sub(qw ones, v47_is_final ));
v83 lacp.vu = spu.sel( v83 lacp.vu, v83 cp.vu, /*pattern*/ spu.cmpeq( spu.and( qw03_next, qw_BIT_SAVE), qw_zeroses ) );
v47 lacp.vu = spu.sel( v47 lacp.vu, v47 cp.vu, /*pattern*/ spu.cmpeq( spu.and( qw47_next, qw_BIT_SAVE), qw_zeroses ) );

qw not at end mask32 = spu.or( spu.cmpeq(v03 buf.end.vu, v03 cp.vu), spu.cmpeq(v47 buf.end.vu, v47 cp.vu) );

const vector unsigned v83_is_final_mask32 = spu.cmpeq(v83_is_final, qw_zeroses);
const vector unsigned v47_is_final_mask32 = spu.cmpeq(v47_is_final, qw_zeroses);
v83 state.vu = qw83_next; v83 state.vu = spu.and( v83 state.vu, qw KILL FLAGS );
v47 state.vu = qw47_next; v47 state.vu = spu.and( v47 state.vu, qw KILL FLAGS );

const vector unsigned v83_is_restore_mask32 = spu.cmpeq( spu.and( qw03_next, qw BIT RESTORE), qw_zeroses );
const vector unsigned v47_is_restore_mask32 = spu.cmpeq( spu.and( qw47_next, qw BIT RESTORE), qw_zeroses );
v83 cp.vu = spu.sel( v83 cp.vu, v83 lacp.vu, v83 is_restore_mask32 );
v47 cp.vu = spu.sel( v47 cp.vu, v47 lacp.vu, v47 is_restore_mask32 );

const variant t v83_is_acronym = { .vu = spu.and( qw03 next, qw BIT ACRONYM ) };
const variant t v47_is_acronym = { .vu = spu.and( qw47 next, qw_BIT_ACRONYM ) };

in_chr_0_pl = * (v03 cp.ucp[0]); in_chr_4_pl = * (v47 cp.ucp[0]);
in_chr_1_pl = * (v03 cp.ucp[1]); in_chr_5_pl = * (v47 cp.ucp[1]);
in_chr_2_pl = * (v03 cp.ucp[2]); in_chr_6_pl = * (v47 cp.ucp[2]);
in_chr_3_pl = * (v03 cp.ucp[3]); in_chr_7_pl = * (v47 cp.ucp[3]);

const vector unsigned qw_ptt_entry_0 = { v03 bp.u[0], v83 bp.u[0], v83_is_acronym.u[0], 0 }; const vector unsigned qw_ptt_entry_4 = { v47 bp.u[0], v47 bp.u[4], v47 bp.u[8], 0 }
const vector unsigned qw_ptt_entry_1 = { v03 bp.u[1], v83 bp.u[1], v83_is_acronym.u[1], 0 }; const vector unsigned qw_ptt_entry_5 = { v47 bp.u[1], v48 bp.u[5], v48 bp.u[9], 0 }
const vector unsigned qw_ptt_entry_2 = { v03 bp.u[2], v83 bp.u[2], v83_is_acronym.u[2], 0 }; const vector unsigned qw_ptt_entry_6 = { v47 bp.u[2], v48 bp.u[6], v48 bp.u[10], 0 }
const vector unsigned qw_ptt_entry_3 = { v03 bp.u[3], v83 bp.u[3], v83_is_acronym.u[3], 0 }; const vector unsigned qw_ptt_entry_7 = { v47 bp.u[3], v48 bp.u[3], v48 bp.u[11], 0 }

v83 bp.vu = spu.sel( v83 bp.vu, v83 cp.vu, v83 is_final_mask32 );
v47 bp.vu = spu.sel( v47 bp.vu, v47 cp.vu, v47 is_final_mask32 );

si_stqd( qwword ) qw_ptt_entry_8 = (qwword) v03_ptt_cur.vu,
si_stqd( qwword ) qw_ptt_entry_9 = (qwword) v03_ptt_cur.vu,

si_stqd( qwword ) qw_ptt_entry_1 = (qwword) spu.rlqbyte( v83_ptt_cur.vu, 4 ), 8 );
si_stqd( qwword ) qw_ptt_entry_2 = (qwword) spu.rlqbyte( v83_ptt_cur.vu, 8 ), 8 );

si_stqd( qwword ) qw_ptt_entry_3 = (qwword) spu.rlqbyte( v03_ptt_cur.vu, 12 ), 8 );

si_stqd( qwword ) qw_ptt_entry_4 = (qwword) v47_ptt_cur.vu,

si_stqd( qwword ) qw_ptt_entry_5 = (qwword) spu.rlqbyte( v47_ptt_cur.vu, 4 ), 8 );
si_stqd( qwword ) qw_ptt_entry_6 = (qwword) spu.rlqbyte( v47_ptt_cur.vu, 8 ), 8 );

si_stqd( qwword ) qw_ptt_entry_7 = (qwword) v47_ptt_cur.vu,

v83 ptt_cur.vu = spu.add(v83 ptt_cur.vu, v83 is_token_incr16);
v47 ptt_cur.vu = spu.add( v47 ptt_cur.vu, v47 is_token_incr16 );
```
Dual cycle: 5.6%  
Branch miss: 24.6%  
Dependency stalls: 37.2%  
CPI = 2.12

Best-case performance:  
212.4 cycles/char  
15.1 MB/s / SPE  
0.96 Gbps / chip

Typical performance:  
330.5 cycles/char  
9.7 MB/s / SPE  
0.62 Gbps / chip

Dual cycle: 8.1%  
Branch miss: 2.1%  
Dependency stalls: 34.1%  
CPI = 1.34

Best-case performance:  
66.2 cycles/char  
48.4 MB/s / SPE  
3.09 Gbps / chip

Typical performance:  
104.6 cycles/char  
30.6 MB/s / SPE  
1.96 Gbps / chip

Dual cycle: 12.1%  
Branch miss: 0.0%  
Dependency stalls: 37.8%  
CPI = 1.21

Best-case performance:  
49.4 cycles/char  
64.8 MB/s / SPE  
4.15 Gbps / chip

Typical performance:  
78.1 cycles/char  
41.0 MB/s / SPE  
2.62 Gbps / chip

Dual cycle: 32.4%  
Branch miss: 0.0%  
Dependency stalls: 7.4%  
CPI = 0.75

Best-case performance:  
18.6 cycles/char  
172.2 MB/s / SPE  
11.02 Gbps / chip

Typical performance:  
29.4 cycles/char  
108.9 MB/s / SPE  
6.97 Gbps / chip

Best-case performance:  
18.6 cycles/char  
205.3 MB/s / SPE  
13.14 Gbps / chip

Typical performance:  
24.57 cycles/char  
129.9 MB/s / SPE  
8.33 Gbps / chip
SPE organization

Input Streams

SPE 1

Input text buffers (double)

State Transition Table (STT)

Output token table buffers (double)

Output Token Tables

DFA1 DFA2 DFA3 DFA4 DFA5 DFA6 DFA7 DFA8
Chip-level organization
Peak and Typical Performance Attained by Kernel 3 across its optimization steps

- Peak Throughput
- Typical Throughput

Throughput (Gbps)

SIMD

8 engines

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22
# Speed-ups by technique

Assuming Kernel 3, Lucene tokenization rules on Wikipedia data.

- **1.32×** SPE runs the code slightly faster than PPE in the experimental conditions we consider.

- **8.0 ×** Employing 8 replicas of the code, running on SPEs. Assuming perfect load balancing, no additional workload.


- **2.20×** 4-way SIMDization. Difference from theoretical 4x is due to non-vector load/stores, pack/unpack operations.

- **1.44×** Duplication of the engines => dependency stalls disappear, dual issue rate increases.

\[=67.57×\] Your mileage may vary (57 – 81). Professional driver on closed course.
Thank you
Questions welcome
Backup slides
<table>
<thead>
<tr>
<th>Labels</th>
<th>Even Pipeline</th>
<th>Odd Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>552</td>
<td></td>
<td>592: stars1,2, w..., current state</td>
</tr>
<tr>
<td>553</td>
<td>603 and $30, $30</td>
<td>604: nop</td>
</tr>
<tr>
<td>556</td>
<td>605: llhu, memory*32248@h</td>
<td></td>
</tr>
<tr>
<td>557</td>
<td></td>
<td>606: llhu, memory*32248@h</td>
</tr>
<tr>
<td>558</td>
<td></td>
<td></td>
</tr>
<tr>
<td>559</td>
<td>607: slgt, $89, $89</td>
<td></td>
</tr>
<tr>
<td>560</td>
<td></td>
<td></td>
</tr>
<tr>
<td>561</td>
<td>608: nop127</td>
<td></td>
</tr>
<tr>
<td>564</td>
<td></td>
<td></td>
</tr>
<tr>
<td>565</td>
<td>610: llhu, memory*32128@h</td>
<td></td>
</tr>
<tr>
<td>567</td>
<td>612: li11, 32</td>
<td>511: hbr, L166, L159</td>
</tr>
<tr>
<td>568</td>
<td>613: li10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>569</td>
<td>615: shli11, 1</td>
<td></td>
</tr>
<tr>
<td>570</td>
<td></td>
<td></td>
</tr>
<tr>
<td>571</td>
<td></td>
<td></td>
</tr>
<tr>
<td>572</td>
<td></td>
<td>609: brz, $6, L97</td>
</tr>
<tr>
<td>573</td>
<td>616: llhu, memory*32128@h</td>
<td></td>
</tr>
<tr>
<td>574</td>
<td></td>
<td></td>
</tr>
<tr>
<td>575</td>
<td></td>
<td></td>
</tr>
<tr>
<td>576</td>
<td>618: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>577</td>
<td></td>
<td></td>
</tr>
<tr>
<td>578</td>
<td></td>
<td></td>
</tr>
<tr>
<td>579</td>
<td>619: li11, 32</td>
<td>618: nop1, 1</td>
</tr>
<tr>
<td>580</td>
<td></td>
<td></td>
</tr>
<tr>
<td>581</td>
<td></td>
<td></td>
</tr>
<tr>
<td>582</td>
<td></td>
<td></td>
</tr>
<tr>
<td>583</td>
<td>620: li10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>584</td>
<td></td>
<td></td>
</tr>
<tr>
<td>585</td>
<td></td>
<td></td>
</tr>
<tr>
<td>586</td>
<td>621: li10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>587</td>
<td></td>
<td></td>
</tr>
<tr>
<td>588</td>
<td></td>
<td></td>
</tr>
<tr>
<td>589</td>
<td></td>
<td>625: shrli10, memory*32128@l</td>
</tr>
<tr>
<td>590</td>
<td></td>
<td></td>
</tr>
<tr>
<td>591</td>
<td></td>
<td></td>
</tr>
<tr>
<td>592</td>
<td></td>
<td></td>
</tr>
<tr>
<td>593</td>
<td>622: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>594</td>
<td></td>
<td></td>
</tr>
<tr>
<td>595</td>
<td></td>
<td></td>
</tr>
<tr>
<td>596</td>
<td></td>
<td></td>
</tr>
<tr>
<td>597</td>
<td>623: li10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>598</td>
<td></td>
<td></td>
</tr>
<tr>
<td>599</td>
<td></td>
<td></td>
</tr>
<tr>
<td>600</td>
<td></td>
<td></td>
</tr>
<tr>
<td>601</td>
<td></td>
<td></td>
</tr>
<tr>
<td>602</td>
<td></td>
<td>642: slq, $30, 32($10)</td>
</tr>
<tr>
<td>603</td>
<td>624: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>604</td>
<td></td>
<td></td>
</tr>
<tr>
<td>605</td>
<td></td>
<td></td>
</tr>
<tr>
<td>606</td>
<td></td>
<td></td>
</tr>
<tr>
<td>607</td>
<td></td>
<td></td>
</tr>
<tr>
<td>608</td>
<td></td>
<td></td>
</tr>
<tr>
<td>609</td>
<td></td>
<td></td>
</tr>
<tr>
<td>610</td>
<td>625: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>611</td>
<td></td>
<td></td>
</tr>
<tr>
<td>612</td>
<td>626: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>613</td>
<td></td>
<td></td>
</tr>
<tr>
<td>614</td>
<td></td>
<td></td>
</tr>
<tr>
<td>615</td>
<td></td>
<td></td>
</tr>
<tr>
<td>616</td>
<td>627: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>617</td>
<td></td>
<td></td>
</tr>
<tr>
<td>618</td>
<td></td>
<td></td>
</tr>
<tr>
<td>619</td>
<td>628: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>620</td>
<td></td>
<td></td>
</tr>
<tr>
<td>621</td>
<td></td>
<td></td>
</tr>
<tr>
<td>622</td>
<td>629: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>623</td>
<td></td>
<td></td>
</tr>
<tr>
<td>624</td>
<td>630: shli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>625</td>
<td></td>
<td></td>
</tr>
<tr>
<td>626</td>
<td>631: nop27</td>
<td></td>
</tr>
<tr>
<td>627</td>
<td></td>
<td></td>
</tr>
<tr>
<td>628</td>
<td>632: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>629</td>
<td></td>
<td></td>
</tr>
<tr>
<td>630</td>
<td></td>
<td></td>
</tr>
<tr>
<td>631</td>
<td>633: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>632</td>
<td></td>
<td></td>
</tr>
<tr>
<td>633</td>
<td>634: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>634</td>
<td></td>
<td></td>
</tr>
<tr>
<td>635</td>
<td>636: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>636</td>
<td></td>
<td></td>
</tr>
<tr>
<td>637</td>
<td>638: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>638</td>
<td></td>
<td></td>
</tr>
<tr>
<td>639</td>
<td>640: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>640</td>
<td></td>
<td></td>
</tr>
<tr>
<td>641</td>
<td>642: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>642</td>
<td></td>
<td></td>
</tr>
<tr>
<td>643</td>
<td>644: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>644</td>
<td></td>
<td></td>
</tr>
<tr>
<td>645</td>
<td>646: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>646</td>
<td></td>
<td></td>
</tr>
<tr>
<td>647</td>
<td>648: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>648</td>
<td></td>
<td></td>
</tr>
<tr>
<td>649</td>
<td>650: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>650</td>
<td></td>
<td></td>
</tr>
<tr>
<td>651</td>
<td>652: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>652</td>
<td></td>
<td></td>
</tr>
<tr>
<td>653</td>
<td>654: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>654</td>
<td></td>
<td></td>
</tr>
<tr>
<td>655</td>
<td>656: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>656</td>
<td></td>
<td></td>
</tr>
<tr>
<td>657</td>
<td>658: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>658</td>
<td></td>
<td></td>
</tr>
<tr>
<td>659</td>
<td>660: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>660</td>
<td></td>
<td></td>
</tr>
<tr>
<td>661</td>
<td>662: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>662</td>
<td></td>
<td></td>
</tr>
<tr>
<td>663</td>
<td>664: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>664</td>
<td></td>
<td></td>
</tr>
<tr>
<td>665</td>
<td>666: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>666</td>
<td></td>
<td></td>
</tr>
<tr>
<td>667</td>
<td>668: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>668</td>
<td></td>
<td></td>
</tr>
<tr>
<td>669</td>
<td>670: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>670</td>
<td></td>
<td></td>
</tr>
<tr>
<td>671</td>
<td>672: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>672</td>
<td></td>
<td></td>
</tr>
<tr>
<td>673</td>
<td>674: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>674</td>
<td></td>
<td></td>
</tr>
<tr>
<td>675</td>
<td>676: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>676</td>
<td></td>
<td></td>
</tr>
<tr>
<td>677</td>
<td>678: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>678</td>
<td></td>
<td></td>
</tr>
<tr>
<td>679</td>
<td>680: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>680</td>
<td></td>
<td></td>
</tr>
<tr>
<td>681</td>
<td>682: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>682</td>
<td></td>
<td></td>
</tr>
<tr>
<td>683</td>
<td>684: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>684</td>
<td></td>
<td></td>
</tr>
<tr>
<td>685</td>
<td>686: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>686</td>
<td></td>
<td></td>
</tr>
<tr>
<td>687</td>
<td>688: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>688</td>
<td></td>
<td></td>
</tr>
<tr>
<td>689</td>
<td>690: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>690</td>
<td></td>
<td></td>
</tr>
<tr>
<td>691</td>
<td>692: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>693</td>
<td>694: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>694</td>
<td></td>
<td></td>
</tr>
<tr>
<td>695</td>
<td>696: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>697</td>
<td>698: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>698</td>
<td></td>
<td></td>
</tr>
<tr>
<td>699</td>
<td>690: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>690</td>
<td></td>
<td></td>
</tr>
<tr>
<td>691</td>
<td>692: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>692</td>
<td></td>
<td></td>
</tr>
<tr>
<td>693</td>
<td>694: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>694</td>
<td></td>
<td></td>
</tr>
<tr>
<td>695</td>
<td>696: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>696</td>
<td></td>
<td></td>
</tr>
<tr>
<td>697</td>
<td>698: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>698</td>
<td></td>
<td></td>
</tr>
<tr>
<td>699</td>
<td>690: shrli10, memory*32128@l</td>
<td></td>
</tr>
<tr>
<td>690</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
• Flex's machine is not a classic DFA
  – bp, cp pointers
  – ...

... no internationalization resources ...

candidate
Optimizing Flex on the Cell/B.E.

- Flex's machine is not a classic DFA
  - bp, cp pointers
  - longest-of-the-leftmost semantics
  - ...

\[
i[a-z]^*n
\]

internationalization

\[
\begin{align*}
(1,2) &\quad \text{in} \\
(1,6) &\quad \text{intern} \\
(1,11) &\quad \text{internation} \\
(1,20) &\quad \text{internationalization} \\
(9,11) &\quad \text{ion} \\
(9,20) &\quad \text{ionalization} \\
(14,20) &\quad \text{ization} \\
(17,20) &\quad \text{ion}
\end{align*}
\]
Optimizing Flex on the Cell/B.E.

- Flex's machine is not a classic DFA
  - bp, cp pointers
  - longest-of-the-leftmost semantics
  - back-up transitions

```
example grammar

LETTER       [A-Za-z]
DIGIT        [0-9]
ALPHA        {LETTER}+
TOKEN        ({LETTER}|{DIGIT})+
ACRONYM      {ALPHA}"."({ALPHA}".")+
%%           /*do nothing*/;
{TOKEN}       printf("Token: %s\n", yytext);
{ACRONYM}     printf("Acronym: %s\n", yytext);
```

```
"a.b.c"
```

(mind the space)
example grammar

```
LETTER   [A-Za-z]
DIGIT    [0-9]
ALPHA    {LETTER}+
TOKEN    ({LETTER}|{DIGIT})+
ACRONYM  {ALPHA}"."({ALPHA}".")+
%%
. | \n    /* do nothing*/;
{TOKEN}  printf("Token: %s\n", yytext);
{ACRONYM} printf("Acronym: %s\n", yytext);
```

"a.b.c"

up to here, acronym candidate (string is a valid acronym prefix)

a minimalistic grammar which causes back-ups

a minimalistic input which causes back-ups
example grammar

```
LETTER   [A-Za-z]
DIGIT    [0-9]
ALPHA    {LETTER}+
TOKEN    ({{LETTER}\|{DIGIT}})+
ACRONYM  {ALPHA}".\"{{ALPHA}".\"}+
\%
.\|\n     /*do nothing*/;
{TOKEN}  printf("Token: %s\n", yytext);
{ACRONYM} printf("Acronym: %s\n", yytext);
```

"a.b.c "
up to here, acronym candidate (string is a valid acronym prefix)

"a.b.c "
automaton processes the space, string cannot be an acronym, back-up!

"a.b.c "
back-up to the last valid matched token
  "a.b." is matched, c will be processed again
Classic DFA

$$(\Sigma, Q, \delta, q_0, F)$$

$s_0 = q_0$

$\delta: Q \times \Sigma \rightarrow Q$

$s_i = \delta(s_{i-1}, I_i)$

result is $s_{||} \in F$,

Advantage:
one transition per input symbol
(DFA-base AC is classic a DFA)
Classic DFA

\((\Sigma, Q, \delta, q_0, F)\)

\(s_0 = q_0\)

\(\delta: Q \times \Sigma \rightarrow Q\)

\(s_i = \delta(s_{i-1}, I_i)\)

result is \(s_{||} \in F\),

Flex's automaton

\((\Sigma, Q, A, \Delta, q_0)\)

\(s_0 = q_0\)

\(\Delta: Q \times \Sigma \rightarrow Q \times \{r, f, s, b\} \times (A \cup \{\})\)

\(\delta(s_{i-1}, I_i) = (s_i, k_i, a_i)\)

emit \(a_i\)

4 kinds of transitions: regular, final, save, backup
Classic DFA

\[(\Sigma, Q, \delta, q_0, F)\]

\[s_0 = q_0\]

\[\delta: Q \times \Sigma \rightarrow Q\]

\[s_i = \delta(s_{i-1}, I_i)\]

result is \(s_{||} \in F\),

Flex's automaton

\[(\Sigma, Q, A, \Delta, q_0)\]

\[s_0 = q_0\]

\[\Delta: Q \times \Sigma \rightarrow Q \times \{r, f, s, b\} \times (A \cup \{\})\]

\[\delta(s_{i-1}, I_i) = (s_i, k_i, a_i)\]

emit \(a_i\)

4 kinds of transitions:
regular, final, save, backup

Backup transitions:
- bad! re-process input multiple times
- how do they work?
LETTER  [A-Za-z]
DIGIT    [0-9]
ALPHA    {LETTER}+
TOKEN    ({LETTER}|{DIGIT})+
ACRONYM  {ALPHA}"."({ALPHA}".")+
%
/*do nothing*/;
{TOKEN}  printf("Token: %s\n", yytext);
{ACRONYM} printf("Acronym: %s\n", yytext);

<table>
<thead>
<tr>
<th>Step</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>a</td>
<td>.</td>
<td>b</td>
<td>.</td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Transition kind</td>
<td>save reg.</td>
<td>reg.</td>
<td>save reg.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Next state</td>
<td>1</td>
<td>6</td>
<td>8</td>
<td>10</td>
<td>11</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>Last saved (cp, a)</td>
<td>–</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(1,1)</td>
<td>(4,3)</td>
<td>(4,3)</td>
<td></td>
</tr>
</tbody>
</table>
LETTER  [A-Za-z]
DIGIT    [0-9]
ALPHA    {LETTER}+
TOKEN    ({LETTER}|{DIGIT})+
ACRONYM  {ALPHA}"."({ALPHA}".")+

/*do nothing*/;
{TOKEN}  printf("Token: %s\n", yytext);
{ACRONYM} printf("Acronym: %s\n", yytext);

Step      0  1  2  3  4  5  6
Input     a . b . c (space)
Transition kind      save reg. reg. save reg. back-up
Next state    1  6  8  10  11  10  1
Last saved (cp, a)  –  (1,1) (1,1) (1,1) (4,3) (4,3) (4,3)

match “a.b.”,
“c” will be processed again
Optimizing Flex on the Cell/B.E.

- Flex's machine is not a classic DFA
  - bp, cp pointers
  - longest-of-the leftmost semantics
  - back-up transitions

- Its sequential implementation is inefficient
  - irregular control flow
  - multiple tables

- Optimization
  - transform control flow in data flow
  - use a single table
  - make transitions cheap
  - ...
next_state = table[current_state][input]
next_state = table[current_state][input]

```
table_base + current_state * row_size + input * cell_size
```

ADD

SHL

ADD

SHL
table_base + current_state * row_size + input * cell_size

ADD          SHL          ADD          SHL

state_code: 16 bit,
cell_size = 2
x*2 = SHL x,1

- Idea: pre-shift input!
- Assumption: input is ASCII (0..127)
  MSB unused, can be shifted in place
  no packing/unpacking!
- In-place SHL1 is naturally 16-way SIMD
- Easy to optimize,
- Unroll 8x, CPI = 0.75
- Throughput: 226.7 Gbps per socket
  0.23 cycles/char
Use (current_state * row_size) as a state identifier!
Just need to re-encode table (compile time)
Constraint: table must be < 64k

```
0 2 4 6 8 ...
0 256 512 768 1024 ...
```

```
0
256
512
768
1024
1280
```
(table_base + current_state_ofs) is actually the state row pointer

Idea: use it as a state identifier!

This can be done only if all the pointers <64k (a cell is only 2 bytes)

Must:
1) instruct linker to place table <64k (custom ld script)
2) relocate table at runtime (quick, once)
2 Main results:

- we got rid of 3 out of 4 operations, we reduced a state transition to an ADD!

- state IDs (= state ptrs) are aligned
  => only the higher 8 bits are used
  we have the lower 8 bits for flags!
Next challenge:

how to generate the output without branches?

Token table (aligned)

<table>
<thead>
<tr>
<th></th>
<th>start</th>
<th>stop</th>
<th>token type</th>
<th>unused</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>start</td>
<td>stop</td>
<td></td>
<td>unused</td>
</tr>
<tr>
<td>32</td>
<td>start</td>
<td>stop</td>
<td>token type</td>
<td>unused</td>
</tr>
<tr>
<td>64</td>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
</tr>
</tbody>
</table>
- Idea: speculative write (software speculation, not hardware)
regular
cp++
save
  save (cp,a)
cp++
final
  match bp...cp-1, action a
  s = q0
  bp = cp
backup
  recall (cp,a)
  match bp...cp, action a
  s = q0
cp++
regular
  cp++
save
  save (cp,a)
  cp++
final
  match bp...cp−1, action a
  s = q0
  bp = cp
backup
  recall (cp,a)
  match bp...cp, action a
  s = q0
  cp++
regular
  cp++

save
  save cp
  cp++

final
  match bp..cp-1, action a
  s = q0
  bp = cp

backup
  recall cp
  match bp..cp, action a
  s = q0
  cp++
Lucene:

original tokenizer 159 states
transformed tokenizer 173 states (+9%)

(5 semantic actions, including ignore)