INTERNATIONAL COMPUTER SCIENCE INSTITUTE

1947 Center St. • Suite 600 • Berkeley, California 94704-1198 • (510) 643-9153 • FAX (510) 643-7684

Some MPEG Decoding Functions on Spert An Example for Assembly Programmers

Arno Formella

TR-94–027

October 1994

Abstract

We describe our method how to implement C–program sequences in torrent (T0) assembler code while there is no efficient automatic tool. We use re-structuring of the source code, vectorization, dataflow graphs, a simple scheduling strategy and a straight forward register allocation algorithm. We define some lower and an upper bound for the expected run time. For two functions, namely the color transformation and reverse DCT, we achieve almost 54, respectively 16 times the performance of a Sparc 2 workstation.

Contents

1 Introduction

The most recent documentation (or at least pointer to that information) can be gathered via WWW–hypertext at ICSI. While writing this technical report the WWW–pages contained the following list of documents being related to T0 and Spert:
– Torrent Architecture Manual (hypertext)
– Torrent Architecture Manual (postscript)
– T0 Engineering data (postscript)
– Diagram of the structure of T0
– Diagrams of the fixed point add and multiply pipes
– A description of the fixed point pipe operations
– A document describing the interface between the host and the SPERT board
– List of opcodes accepted by "toras", the SPERT assembler
– Details of SPERT simulators
– Info files for "gas", the GNU assembler (on which toras is based)
– Info files for "gcc", the GNU C/C++ compiler used with SPERT.

T0 is a processor with the following main properties:

- MIPS II compatible instruction set
- sixteen 32–Bit wide and vector registers of length 32
- three SIMD functional blocks: two ALUs consisting of eight units and one multiplier consisting of eight units
- the functional units are fully pipelined and can work in parallel on the register file, thus a vector operation needs at most 4 clock cycle to be executed.
- chaining of operations is possible
- some special opcodes are implemented (such as shift&add, and conditional merge)

Spert is a board equipped with one T0 and memory which can be used in a workstation.

In the following sections, we describe how to write quite efficient assembly code for T0, while there is still no automatic tool available. We start from an original C–source code (some functions of an MPEG decoder). We transform the code into a more suiteable form for the processor, especially vectorization is performed. After that we construct the dataflow graphs and calculate some lower and upper bounds for the run time. A simple way is introduced how to handle register allocation and instruction scheduling by hand. On the example of the MPEG–functions we use the described methods to show the performance of T0.

2 C–Code

The sources are written in C. We restructure the original source code in the high level language in order to exploit special properties of the processor. Sometimes, it is better not to perform special kinds of optimizations, which are introduced in the code to increase the performance on a workstation:

- Table lookup is replaced by recalculation, because an indirect memory access is more expensive than the additional arithmetic operations.
- Because T0 has more powerful shift instructions which can take place in parallel to other operations, it is not necessary to reduce the number of shift instructions.
- Generally, because T0 has multiple functional units which can work in parallel, one should not merely reduce the number of operations.

3 Vectorizing

We vectorize the innermost loops of the restructured program sequence. For that reason, we introduce a vector extension of C, which is easy to understand, so we don't give a formal introduction. The basic idea is the use of range expressions to denote vector variables. So the line `a[1:n]=b[1:(2)2n]+c[1:n];`
means that the vector consisting of the odd elements of vector `b` is added to vector `c` and stored into vector `a`.

Such a vector expression implies parallel calculations both on the data and the addresses. The algorithm and/or the data strauctures can be modified in such a way that the calculation can really be performed in parallel on the hardware.

Vectorization is a very important factor in increasing the performance of the program sequence. Sometimes, the entire algorithm should be changed in order to keep the vector pipelines busy. Inspite of a possible increase of the number of actually performed operations, the run time can be improved, for instance through recursive doubling, block parallel operations with final merge, and avoiding to deal with special cases).

Classical concepts, such as loop unrolling, can be used in vectorized code as well. The aim of such techniques is to get larger basic blocks, which can be optimized more effectively. However, the limited amount of registers can become a severe problem. Loop unrolling can be necessary in order to maintain an invariant loop condition, so a re–entry in the loop can be done without moving register contents. Special cases, often the first and last iteration of a loop, can be put outside of the loop which makes additional conditional statement inside of the loop obsolet.

Another important fact is the memory map of the data. The data should be stored in such a manner that fast vector reads and writes can be performed. For example transposing matrices at the right time can reduce the run time of the program sequence.

4 Dataflow Graphs

We split the program in basic blocks, which are sequences of instructions without branches. Nevertheless, if `if`–statements are vectorized, they do not count as branch instructions.

The dataflow graph of a basic block is a dag (directed acyclic graph). The nodes represent machine instructions (an extension to sequences of machine instructions, which will not be "atomized" later, is possible). They are labeled with the appropriate instruction. The indegree of a node depends on the type of the instruction the node is labeled with. An indegree of two means binary instruction, an indegree of one means unary instruction,

an indegree of zero occurs for certain load instructions. An indegree of three may occur, if conditional instructions—or multi address operations—are available.

Variables of the original program sequence are added as labels to the edges of the graph. This is done with the names of the variables in brackets.

The nodes are labeled arbitrary with numbers in parenthesis. (A good labeling scheme is from left to right and from top to bottom.) These numbers are used later to identify the instruction in the dag with the corresponding instruction in the assembly program; there, they occur as comments.

Nodes with vector instructions are drawn as circles and nodes with scalar instructions are drawn as squares.

Edges may be labeled with address sequences according to memory, or with register names available in the processor (which can be seen as addresses in a "memory of the processor"). Those addresses or registers are added while scheduling the dataflow graph into machine instructions.

5 Lower and Upper Bounds for the Run Time

In this section we will derive some lower and upper bounds for the run time of an instruction sequence corresponding to a dag. The bounds let us define a mean to measure the efficiency of an optimized instruction sequence.

5.1 Lower Bounds

We define four lower bounds:

- u_m as the number of the cycles while the memory port is busy due to load and store instructions.
- u_a as the number of the cycles due to arithmetical operations, where it is possible that more than one operation is performed in the same cycle, if there are several functional units operating in parallel.
- u_i as the number of instructions (note that loads take normally two or three instructions).
- u_l as the number of the cycles to execute the longest path of the dataflow graph.

Clearly, those bounds depend on the vector length. The maximum of the lower bounds gives an overall lower bound u for the run time, i. e.

$$u = \max(u_m, u_a, u_i, u_l)$$

We call a program sequence memory limited, ALU limited, instruction issue limited or data dependency limited, if the corresponding lower bound u_m, u_a, u_i or u_l, respectively, is much larger than all other bounds.

It seems that a limited program is easier to optimize, although such an optimization might not lead to a great improvement of the run time. Especially a memory limited

program can be optimized by trying to interleave arithmetical operations with load and stores, e. g. by loading in one iteration of a loop all data used in the next iteration. A data dependency limited program cannot be optimized within that basic block; possible optimizations must be searched while merging basic blocks. If a program is instruction issue limited, the only difficulty lies in finding a schedule of the instructions which avoid idle cycles, but this can be done straight forward, because there are not many data dependencies. More challanging cases are those, where the program is ALU limited. One must find a good schedule of the operations, so all arithmetical pipes can be held busy.

The most difficult cases are those where all of the lower bounds are in the same range. Here many constraints make it difficult to find a good or optimal schedule and it is not clear if the run time in the end is close to the lower bounds.

5.2 Upper Bounds

In order to write code for the evaluation of a dag one can always use the simple way of coding the instructions in any topological order. In almost all cases, this will not lead to an optimal run time, but it gives an upper bound for the runtime.

If there are no memory accesses with stride addresses or index addresses and all instructions reside in the cache we calculate the upper bound of the run time as follows:

- If the length of the vectors is less or equal to sixteen then at most two idle cycles are possible after each instruction issue, which may be caused by interlocks, such as a register is read directly after it has been written. Thus, an upper bound v is given by three times the number of instructions.

- If the length of the vectors is greater than sixteen three idle cycles may be inserted, because the of unavailability of functional units. Thus, an upper bound v is given by four times the number of instructions.

If there are memory accesses with stride addresses or index addresses, we add the access time to memory to the previous described bound and obtain the upper bound for those cases.

Because even in a random schedule of instructions certain overlaps of the execution time of operations are very likely, an upper bound of two times the number of instructions plus the additional idle cycles due to stride/indexed load or stores can be taken as a rough estimation.

Remark: If it is quite certain that the instructions are not located in the instruction cache of T0, the lower bound due to instruction issue u_i is inceased by a factor of about two, because all instructions must be taken out of memory. Therefore, if a program sequence is instruction issue limited and not encached, it is not worth while to optimize this sequence, because u_i is almost equal to v.

5.3 Efficiency and Improvement

The discussion of upper bounds shows, that an optimization through scheduling of the operations of a dataflow graph can give at most an improvement of about a factor of two

to three compared to a random schedule. We measure the *efficiency* η of a schedule with the quotient of the maximal lower bound u and the final run time t of the schedule

$$\eta = u/t$$

The minimal efficiency η_{min}, i. e. the worst what one can expect, is given by the quotient of the upper and lower bound:

$$\eta_{min} = u/v$$

The reciprocal value of the minimal efficiency is called *possible improvement* q,

$$q_{max} = 1/\eta_{min}$$

which means that the run time of an optimized program sequence can be at most q times faster than the run time of a naive schedule. It might not be possible to achieve such a high improvement, because the constraints introduced through the data dependencies do not allow for avoiding all idle cycles. Therefore, we call

$$q = 1/\eta$$

achieved improvement. Note that the efficiency is not a mean to decide which vector length is the best to be chosen.

6 Register Allocation

We assume that at the beginning of the evaluation of the dag all operands are located in memory and that the results are stored in memory too. Exchanging parameters through registers is not discussed here, because this would require a general mechanism for the transfer of parameters in a compiler.

All registers are free to be used in the evaluation of the dag. We perform the following simple algorithm to handle the registers. The registers are held in a list, which is ordered according to the numbers of the registers. If a new register is needed, the register with the lowest number is taken out of the list. If a register is not used any more, it is put back into the list.

If the list is empty, i. e. all registers are in use, but another register is required, we extend the list through a virtual register. After the assembly program has been completed, we add spill code, i. e. for each usage of a virtual register we try to find the real register which induces the smallest loss in run time through spilling. We consider that the store and later the load instruction should be performed in a time slot where the memory pipe is not busy.

7 Assembly Program

We add an outer loop to the basic block. The body of the loop is executed two times: the first time we get the non–encached run time, the second time we get the encached run time.

The following is a list of techniques an heuristics we used to schedule the dataflow graph into machine instruction with the aim to avoid idle cycles of the processor.

- Load and store instructions are separated as far as possible.
- First we just implement the instructions without register specifications. Those are added in a second path using the method described above. Because we assume in the first path that there are enough registers available, additional loads and stores for intermediate results are added after the program has been scheduled. This may result in non–optimal schedules, but we beleave it is a good heuristic.
- We start with the longest path in the dataflow graph. This path is implemented as instruction with additional `nop`–instructions which represent the necessary delays (due to chaining, availability of functional units, or load and stores idle cycles).
- We add the rest of the nodes for one result of the dag, before considering other terminal nodes of the dag.
- Doing this, we try to separate multiplications (because the processor has only one functional unit performing multiplications) and we try to fill in independent instructions after loads and stores to use the idle cycles.
- If the dag constist of two or more main paths, we try to merge those paths in a way that the instructions in the paths are executed alternately.
- Scalar (integer) operations are scheduled after all vector operations have been scheduled, because it is likely, that idle cycle can be found where those—often one cycle—instructions fit perfectly.

8 Run Time

We obtain the run time in clock cycles cc with the RTL–simulator for T0. All run times in ms, we give in the following, are based on a 40 MHz version of T0.

9 MPEG–Decoder

We took the program code for MPEG–decoding available via anonymous ftp from UCB at site: `toe.cs.berkeley.edu` ([Le 91], [PSR93]). Profiling the decoding part on one test film gives the run time of the different functions. The following table shows where most of the time is spent:

```
  %  cumulative    self              self    total
time   seconds   seconds    calls  ms/call  ms/call name
29.6      3.87     3.87      121    31.98    31.98  _ColorDitherImage [8]
19.5      6.42     2.55    49798     0.05     0.05  _j_rev_dct [9]
13.7      8.22     1.80                             mcount (183)
12.2      9.82     1.60    58080     0.03     0.03  _ReconIMBlock [10]
11.8     11.36     1.54    58080     0.03     0.09  _ParseReconBlock [5]
 6.0     12.15     0.79   851129     0.00     0.00  .umul [11]
```

The time for `mcount` is lost due to profiling, thus 34.3% are spent in `ColorDitherImage`, 22.6% in `j_rev_dct` and 27.8% in both `Block`–Functions, which covers 84.7% of the total run time. Note that the test scene constists only of I–frames. There are no interpolating B– or P–frames between I–frames. The profile was taken on a Sparc 2.

In the following we will program the functions `ColorDitherImage` and `j_rev_dct` for T0.

9.1 Dithering

This section deals with the function `ColorDitherImage()`, which converts the image representation from *YCrCb–space* to *rgb–space*. In *YCrCb–space* each pixel is encoded with one lumination value, but four pixel in a block share two chromatic values. This encoding reduces the image size to half of an encoding in *rgb–space*.

9.1.1 C–Code

The code for a workstation processor contains a lookup table to speedup some calculations. On T0 the usage of this lookup table would require indirect memory accesses. If we compare the time spent in the calculation and the time needed for a memory access, we see that such a lookup table is not a great help for T0; it is better to recalculate the required values. The original C–code looks like:

```
#define BITS            8
#define ONE             ((int) 1)
#define CONST_SCALE     (ONE << BITS)
#define UP(x)           (((int)(x)) << BITS)
#define FIX(x)          ((int) ((x)*CONST_SCALE+0.5))
#define CLAMP(ll,x,ul)  (((x)<(ll)) ?(ll):( ((x)>(ul)) ?(ul):(x)))

static int *Cb_r_tab, *Cr_g_tab, *Cb_g_tab, *Cr_b_tab;

/*
 * InitColorDither --
 *   To get rid of the multiply and other conversions in color
 *   dither, we use a lookup table.
 */
void InitColorDither()
{
    int CR, CB, i;

    Cr_b_tab=(int *)malloc(256*sizeof(int));
    Cr_g_tab=(int *)malloc(256*sizeof(int));
    Cb_g_tab=(int *)malloc(256*sizeof(int));
    Cb_r_tab=(int *)malloc(256*sizeof(int));

    for (i=0; i<256; i++) {
        CB=CR=i-128;
        Cb_r_tab[i]= FIX(1.40200)*CB;  Cr_g_tab[i]=-FIX(0.34414)*CR;
        Cb_g_tab[i]=-FIX(0.71414)*CB;  Cr_b_tab[i]= FIX(1.77200)*CR;
    }
}
```

The following code segment performs the conversion from one color space into the other color space. We give some explanations:

- Two nested `for`–loops run over the entire image, the outer one steps over the rows, the inner one over the columns.

- Because in *YCrCb–space* the color matrices have half the width and height of the lumination matrix, the inner block handles four pixels.
- First the color indices are loaded. They are used for a lookup in the color table.
- The color value and the lumination value are combined and clamped to the precision required in *rgb–space*.

```
/*
 * ColorDitherImage --  Converts image into 24 bit color.
 */
void ColorDitherImage(lum, cr, cb, out, rows, cols)
  unsigned char *lum, *cr, *cb, *out;
  int cols, rows;
{
    int           L, CR, CB, R, G, B;
    unsigned int  *row1, *row2, r, b, g;
    unsigned char *lum2;
    int           x, y, cb_r, cr_g, cb_g, cr_b;

    row1=(unsigned int *)out;  row2=row1+cols;
    lum2=lum+cols;
    for (y=0; y<rows; y+=2) {
        for (x=0; x<cols; x+=2) {

            CR=*cr++;  CB=*cb++;
            cb_r=Cb_r_tab[CB];  cr_g=Cr_g_tab[CR];
            cb_g=Cb_g_tab[CB];  cr_b=Cr_b_tab[CR];

            L=*lum++; L=UP(L);
            R=L+cb_r; G=L+cr_g+cb_g; B=L+cr_b;

            r= CLAMP(0,R,UP(255)) >> BITS;
            g= CLAMP(0,G,UP(255)) & 0xff00;
            b=(CLAMP(0,B,UP(255)) & 0xff00) << BITS;

            *row1++=r|g|b;

            L=*lum++; L=UP(L);
            R=L+cb_r; G=L+cr_g+cb_g; B=L+cr_b;

            r= CLAMP(0,R,UP(255)) >> BITS;
            g= CLAMP(0,G,UP(255)) & 0xff00;
            b=(CLAMP(0,B,UP(255)) & 0xff00) << BITS;

            *row1++=r|g|b;

            /* Now, do second row. */

            L=*lum2++; L=UP(L);
            R=L+cb_r;  G=L+cr_g+cb_g; B=L+cr_b;

            r= CLAMP(0,R,UP(255)) >> BITS;
            g= CLAMP(0,G,UP(255)) & 0xff00;
            b=(CLAMP(0,B,UP(255)) & 0xff00) << BITS;
            *row2++=r|g|b;

            L=*lum2++; L=UP(L);
            R=L+cb_r;  G=L+cr_g+cb_g; B=L+cr_b;

            r= CLAMP(0,R,UP(255)) >> BITS;
            g= CLAMP(0,G,UP(255)) & 0xff00;
            b=(CLAMP(0,B,UP(255)) & 0xff00) << BITS;
            *row2++=r|g|b;
        }
        lum+=cols; lum2+=cols; row1+=cols; row2+=cols;
    }
}
```

The above code was written in order to reduce the number of shift instructions. T0 is able to perform shift instructions in parallel with arithmetical instructions. Therefore, we have rewritten the code in order to avoid the AND–instructions and we have rewritten the expression such that arithmetical and shift operations are close to each other. Further, we have modified the use of the variables in order to reduce the number of them (which probably is done by the compiler in the latter case) and we have replaced the access to the lookup table with recalculation code.

```
#define MAX     0xff
#define BITS2 2*BITS

/*
 * ColorDitherImage --  rewritten,   Converts image into 24 bit color.
 */
void ColorDitherImage(lum, cr, cb, out, rows, cols)
  unsigned char *lum, *cr, *cb, *out;
  int cols, rows;
{
    int           L, CR, CB;
    unsigned int  *row1, *row2, r, b, g;
    unsigned char *lum2;
    int           x, y, c_r, c_g, c_b;

    row1=(unsigned int *)out;
    row2=row1+cols;
    lum2=lum+cols;
    for (y=0; y<rows; y+=2) {
        for (x=0; x<cols; x+=2) {

            CR=*cr++;  CB=*cb++;
            c_r = FIX(1.40200)*CB;  c_g =-FIX(0.71414)*CB;
            c_g+=-FIX(0.34414)*CR;  c_b = FIX(1.77200)*CR;

            L=*lum++; L=UP(L);
            r=(L+c_r)>>BITS;   g=(L+c_g)>>BITS;   b=(L+c_b)>>BITS;
            r=CLAMP(0,r,MAX);  g=CLAMP(0,g,MAX);  b=CLAMP(0,b,MAX);

            *row1++=r|(g<<BITS)|(b<<BITS2);

            L=*lum++; L=UP(L);
            r=(L+c_r)>>BITS;   g=(L+c_g)>>BITS;   b=(L+c_b)>>BITS;
            r=CLAMP(0,r,MAX);  g=CLAMP(0,g,MAX);  b=CLAMP(0,b,MAX);

            *row1++=r|(g<<BITS)|(b<<BITS2);

            /* Now, do second row. */

            L=*lum2++; L=UP(L);
            r=(L+c_r)>>BITS;   g=(L+c_g)>>BITS;   b=(L+c_b)>>BITS;
            r=CLAMP(0,r,MAX);  g=CLAMP(0,g,MAX);  b=CLAMP(0,b,MAX);

            *row2++=r|(g<<BITS)|(b<<BITS2);

            L=*lum2++; L=UP(L);
            r=(L+c_r)>>BITS;   g=(L+c_g)>>BITS;   b=(L+c_b)>>BITS;
            r=CLAMP(0,r,MAX);  g=CLAMP(0,g,MAX);  b=CLAMP(0,b,MAX);

            *row2++=r|(g<<BITS)|(b<<BITS2);
        }
        lum+=cols; lum2+=cols; row1+=cols; row2+=cols;
    }
}
```

9.1.2 Vectorization

We vectorize the inner loop over the rows. Two rows are calculated within the resulting inner block. Because the vectors start with index 1, we have modified the loop bounderies. The UP–shift

operations on lumination vector `L` have been integrated into the expressions for the rgb–vectors, because such an entire expression can be evaluated with a single machine instruction.

Access of memory using stride addressing is expensive. Therefore, we overlap such load and store instructions with independent arithmetical operations. We have taken the first access to the lumination vector and some calculations (parts of the replacement code for the table lookup) outside of the loop, so we have as starting condition of the loop: one lumination vector is loaded and parts of the color calculations are done. If we enter the loop, the calculations for the loaded vector can be finished – all input data is available, and we can load the next lumination vector such that an overlap of independent operations is possible. In order to maintain the loop invariant according to register contents, we have to unroll the loop again. In other words, basic parts of the scheduling of operations is made in the high level code. The `if`–statements in the loop provide that no vectors are loaded in the last iteration of the loop, which will not be used later.

```
/*
 * ColorDitherImage --  vectorized,  Converts image into 24 bit color.
 */
void ColorDitherImage(V_lum, V_cr, V_cb, V_out, rows, cols)
  unsigned int V_lum[1:rows][1:cols], V_out[1:rows][1:cols];
  unsigned char V_cr[1:rows/2][1:cols/2], V_cb[1:rows/2][1:cols/2];
  int cols, rows;
{
    int          V_L1[1:cols/2], V_L2[1:cols/2], V_L3[1:cols/2], V_L4[1:cols/2];
    int          V_CR[1:cols/2], V_CB[1:cols/2];
    int          y;
    unsigned_int V_r[1:cols/2], V_b[1:cols/2], V_g[1:cols/2];
    int          V_c_r1[1:cols/2], V_c_g1[1:cols/2], V_c_b1[1:cols/2];
    int          V_c_r2[1:cols/2], V_c_g2[1:cols/2], V_c_b2[1:cols/2];

    V_L1[1:cols/2] =V_lum[1][1:(2)cols];
    V_CR[1:cols/2] =V_cr[1][1:cols/2]-128;
    V_c_b1[1:cols/2]=FIX(1.77200)*V_CR[1:cols/2];
    V_CB[1:cols/2] =V_cb[1][1:cols/2]-128;
    V_c_r1[1:cols/2]=FIX(1.40200)*V_CB[1:cols/2];

    for (y=1; y<=rows; y+=4) {

        V_c_g1[1:cols/2] =-FIX(0.71414)*V_CB[1:cols/2];
        V_c_g1[1:cols/2]+=-FIX(0.34414)*V_CR[1:cols/2];

        if (y+4<=rows) {
            V_CR[1:cols/2] =V_cr[y/2+1][1:cols/2]-128;
            V_c_b2[1:cols/2]=FIX(1.77200)*V_CR[1:cols/2];
            V_CB[1:cols/2] =V_cb[y/2+1][1:cols/2]-128;
            V_c_r2[1:cols/2]=FIX(1.40200)*V_CB[1:cols/2];
        }

        V_r[1:cols/2]=(UP(V1_L[1:cols/2])+V_c_r1[1:cols/2])>>BITS;
        V_g[1:cols/2]=(UP(V1_L[1:cols/2])+V_c_g1[1:cols/2])>>BITS;
        V_b[1:cols/2]=(UP(V1_L[1:cols/2])+V_c_b1[1:cols/2])>>BITS;

        V_L2[1:cols/2]=V_lum[y][2:(2)cols];

        V_r[1:cols/2]=CLAMP(0,V_r[1:cols/2],MAX);
        V_g[1:cols/2]=CLAMP(0,V_g[1:cols/2],MAX);
        V_b[1:cols/2]=CLAMP(0,V_b[1:cols/2],MAX);

        V_row[y/2][1:(2)cols]=
              V_r[1:cols/2]+(V_g[1:cols/2]<<BITS)+(V_b[1:cols/2]<<BITS2);

        V_r[1:cols/2]=(UP(V2_L[1:cols/2])+V_c_r1[1:cols/2])>>BITS;
        V_g[1:cols/2]=(UP(V2_L[1:cols/2])+V_c_g1[1:cols/2])>>BITS;
        V_b[1:cols/2]=(UP(V2_L[1:cols/2])+V_c_b1[1:cols/2])>>BITS;

        V_L3[1:cols/2]=V_lum[y+1][1:(2)cols];
```

```
        V_r[1:cols/2]=CLAMP(0,V_r[1:cols/2],MAX);
        V_g[1:cols/2]=CLAMP(0,V_g[1:cols/2],MAX);
        V_b[1:cols/2]=CLAMP(0,V_b[1:cols/2],MAX);

        V_row[y/2][2:(2)cols]=
              V_r[1:cols/2]+(V_g[1:cols/2]<<BITS)+(V_b[1:cols/2]<<BITS2);

            /* Now, do second row. */

        V_r[1:cols/2]=(UP(V3_L[1:cols/2])+V_c_r1[1:cols/2])>>BITS;
        V_g[1:cols/2]=(UP(V3_L[1:cols/2])+V_c_g1[1:cols/2])>>BITS;
        V_b[1:cols/2]=(UP(V3_L[1:cols/2])+V_c_b1[1:cols/2])>>BITS;

        V_L4[1:cols/2]=V_lum[y+1][2:(2)cols];

        V_r[1:cols/2]=CLAMP(0,V_r[1:cols/2],MAX);
        V_g[1:cols/2]=CLAMP(0,V_g[1:cols/2],MAX);
        V_b[1:cols/2]=CLAMP(0,V_b[1:cols/2],MAX);

        V_row[y/2+1][1:(2)cols]=
              V_r[1:cols/2]+(V_g[1:cols/2]<<BITS)+(V_b[1:cols/2]<<BITS2);

        V_r[1:cols/2]=(UP(V4_L[1:cols/2])+V_c_r1[1:cols/2])>>BITS;
        V_g[1:cols/2]=(UP(V4_L[1:cols/2])+V_c_g1[1:cols/2])>>BITS;
        V_b[1:cols/2]=(UP(V4_L[1:cols/2])+V_c_b1[1:cols/2])>>BITS;

        if(y+4<=rows) V_L1[1:cols/2]=V_lum[y+2][1:(2)cols];

        V_r[1:cols/2]=CLAMP(0,V_r[1:cols/2],MAX);
        V_g[1:cols/2]=CLAMP(0,V_g[1:cols/2],MAX);
        V_b[1:cols/2]=CLAMP(0,V_b[1:cols/2],MAX);

        V_row[y/2+1][2:(2)cols]=
              V_r[1:cols/2]+(V_g[1:cols/2]<<BITS)+(V_b[1:cols/2]<<BITS2);

        /* analog code with exchanged V_c_(rgb)1 and V_c_(rgb)2 */
        ...
    }
}
```

9.1.3 Dataflow Graphs

From the vector code above we can construct the dataflow graph. We will not draw the whole graph for the `for`–loop. But it can be formed by putting the following parts together:

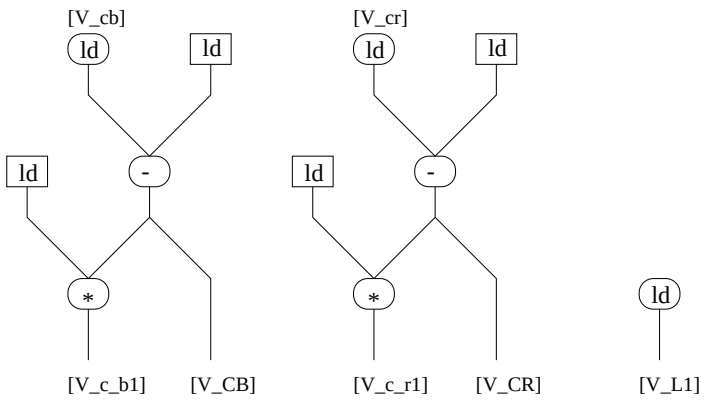

Figure 1: Data flow graph for the code outside the `for`–loop and for the `if`–statements.

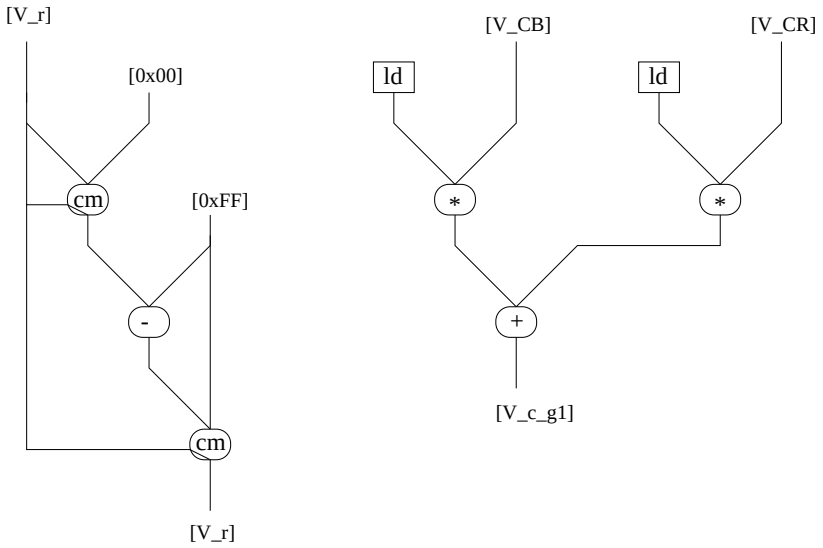

Figure 2: Data flow graph for `CLAMP`–macro and beginning of `for`–loop. (`cm` stands for conditional move instruction.)

9.1.4 Lower and Upper Bounds for the Run Time

We give the bounds for four rows each one constisting of 64 pixel. This amount of data is handled within one pass of the `for`–loop. We assume that all vector operations are performed with vectors of length 32. We consider two cases: a normal case, where the lumination matrix is stored as given in the original program; and the interleaved case, where we assume that the columns of the lumination matrix are rearranged such that first the even then the odd columns are read through row wise accesses.

In the normal case, four vectors are loaded with standard addressing, eight vectors are loaded with constant stride addressing and eight vectors are stored with constant stride addressing. Thus, the lower bound due to memory access is $u_m = 528$ *cc*. In the interleaved case, reading of the lumination vectors can be done with standard addressing, which reduces the access time to $u_m = 304$ *cc*.

We assume that the six scalar values are loaded before entering the `for`–loop and that they are held in register. There are twelve vector load instructions and eight vector store instructions. Each of them is implemented with two instructions. There are 126 arithmetical operations, each counts one instruction. Thus, the lower bound due to instructions is $u_i = 166$ *cc*.

There are less multiplications than other operations. All functional units can work in parallel. Thus, the lower bound due to availability of the functional units is equal to the total number of arithmetical operations divided by the number of functional units and multiplied by the time to deal with one vector. So we have $u_a = 252$ *cc* as lower bound.

The longest path is given by loading a lumination vector, performing the main calculation and storing the result. This leads to a lower bound of $u_l = 81$ *cc* for the normal case. In the interleaved case, the longest path is not determined by loading the lumination vector. Here, the longest path is due to performing the recalculation code, the main calculation and storing of the result, which adds up to $u_l = 62$ *cc*.

The upper bound—vectors have length 32—is four times the number of instructions plus the time for the strided memory accesses, thus $v = 1176$ *cc* for the normal case and $v = 920$ *cc* for the interleaved case.

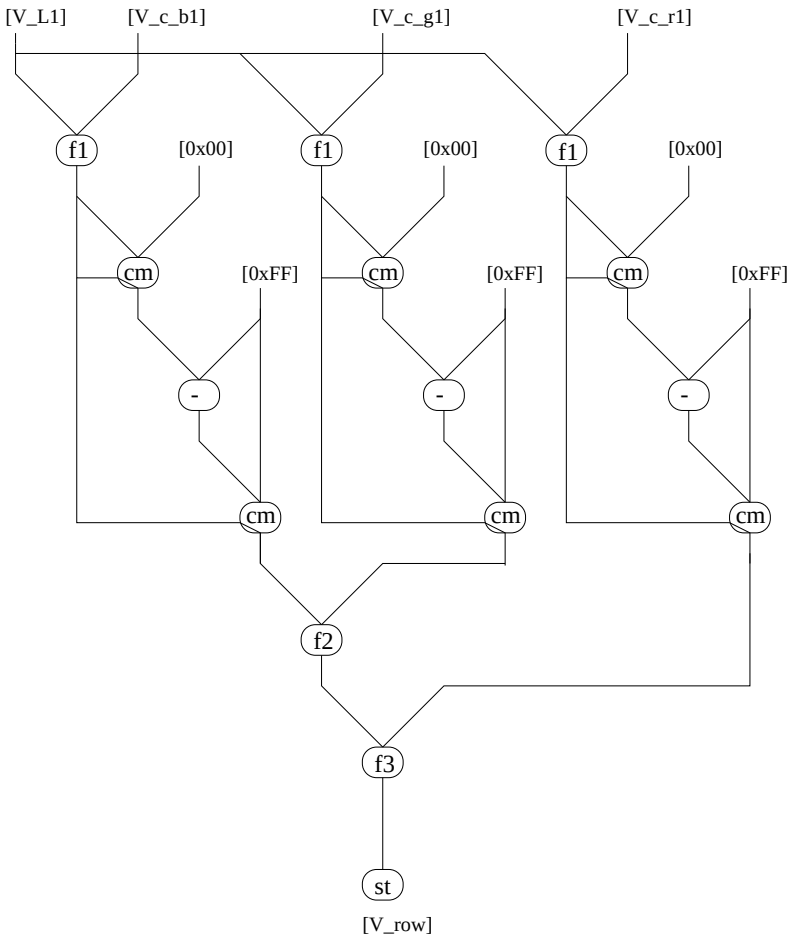

Figure 3: Data flow graph for the main part of the calculation for one row. (`f1`, `f2` and `f3` are the instructions performing simultaneously shift and add operations.)

We see, in both cases the program is memory limited. Table 1 summarizes the bounds.

9.1.5 Register Allocation and Assembly Program

All variables can be loaded before entering the `for`–loop and they can be held in registers. We will not implement the program in assembly code, because it is obvious that the memory access is the bottleneck for the execution time. Even in the interleaved case, access to memory is the most time consuming factor. The dataflow graph consists of many independent parts and the load and store instructions can be scheduled in such a way that probably no idle cycles appear. It seems, that 15 vector registers are sufficient, because only 5 of them are needed to maintain the loop invariant. So, we have 10 registers to evaluate the expressions.

9.1.6 Run Time

We do not consider the initial settings of the vector length register or other constants, such as shift distance or stride distance. Moreover, we assume that the loop management can be handled in idle cycles due to the vector loads as well. The efficiency will be very close to one, because the memory access is the bottleneck. Table 2 shows the run times for the test film.

			memory map	
			normal	interleaved
lower bounds	memory	u_m	528	304
	instructions	u_i	166	166
	operations	u_a	252	252
	dependencies	u_l	81	62
	maximal	u	528	304
upper bound		v	1176	920
minimal efficiency		η_{min}	0.45	0.33
possible improvement		q_{max}	2.22	3.00

Table 1: Lower and upper bounds for the run time of `ColorDitherImage()`

	Sparc2	Spert	
		normal	interleaved
dithering	31.98 *ms*	1.06 *ms*	0.61 *ms*

Table 2: Run time of TO for `ColorDitherImage()` compared to measured run time on Sparc 2

9.2 Reverse DCT

This sections deals with the function `j_rev_dct()`, a reverse decrete cosine transformation.

9.2.1 C–Code

The following are the comments added to the original source code which explain the way the fixed point arithmetic is performed:

> We have to do addition and subtraction of the integer inputs, which is no problem, and multiplication by fractional constants, which is a problem to do in integer arithmetic. We multiply all the constants by `CONST_SCALE` and convert them to integer constants (thus retaining `CONST_BITS` bits of precision in the constants). After doing a multiplication we have to divide the product by `CONST_SCALE`, with proper rounding, to produce the correct output. This division can be done cheaply as a right shift of `CONST_BITS` bits. We postpone shifting as long as possible so that partial sums can be added together with full fractional precision.
>
> The outputs of the first pass are scaled up by `PASS1_BITS` bits so that they are represented to better-than-integral precision. These outputs require `BITS_IN_JSAMPLE + PASS1_BITS + 3` bits; this fits in a 16-bit word with the recommended scaling. (To scale up 12–bit sample data further, an intermediate `INT32` array would be needed.)
>
> In order to avoid overflow of the 32-bit intermediate results in pass 2, we must have `BITS_IN_JSAMPLE + CONST_BITS + PASS1_BITS <= 26`. Error analysis shows that the values given below are the most effective.

Some explanations for a easier understanding of the function:

1. The function takes a 8×8–matrix of 16–bit values as input data. The elements are accessed via pointer `dataptr`. Later we will use `DCTr0`, ..., `DCTr7` as names for the rows and `DCTc0`, ... `DCTc7` as names for the columns.

2. First on the rows then on the columns a 1–dimensional reverse DCT is performed. This is handled through two `for`–loops, which are almost identical.
3. Within each of the two loops first the elements with even index then those with odd index are treated. In the end the partial results are combined.

```
#define CONST_BITS      13
#define DCTSIZE         8
#define PASS1_BITS      2
#define ONE             ((int) 1)
#define CONST_SCALE     (ONE << CONST_BITS)
#define FIX(x)          ((int)((x) * CONST_SCALE+0.5))
#define DESCALE(x,n)    (((x)+(ONE << ((n)-1)) >> (n))

void j_rev_dct (data)
  short data[];
{
  int tmp0, tmp1, tmp2, tmp3, tmp10, tmp11, tmp12, tmp13;
  int z1, z2, z3, z4, z5;
  register short *dataptr;
  int rowctr;

  /* First Part:  rows. */

  dataptr=data;
  for (rowctr=DCTSIZE-1; rowctr >= 0; rowctr--) {
    /* Even part */

    z2=(int) dataptr[2];  z3=(int) dataptr[6];

    z1=z2+z3*FIX(0.541196100);
    tmp2=z1+z3*-FIX(1.847759065);  tmp3=z1+z2*FIX(0.765366865);

    tmp0=((int) dataptr[0]+(int) dataptr[4]) << CONST_BITS;
    tmp1=((int) dataptr[0]-(int) dataptr[4]) << CONST_BITS;

    tmp10=tmp0+tmp3;  tmp13=tmp0-tmp3;  tmp11=tmp1+tmp2;  tmp12=tmp1-tmp2;

    /* Odd part */

    tmp0=(int) dataptr[7];  tmp1=(int) dataptr[5];
    tmp2=(int) dataptr[3];  tmp3=(int) dataptr[1];

    z1=tmp0+tmp3;  z2=tmp1+tmp2;  z3=tmp0+tmp2;  z4=tmp1+tmp3;
    z5=z3+z4*FIX(1.175875602);

    tmp0=tmp0*FIX(0.298631336);  tmp1=tmp1*FIX(2.053119869);
    tmp2=tmp2*FIX(3.072711026);  tmp3=tmp3*FIX(1.501321110);
    z1=z1*-FIX(0.899976223);     z2=z2*-FIX(2.562915447);
    z3=z3*-FIX(1.961570560);     z4=z4*-FIX(0.390180644);

    z3+=z5;  z4+=z5;

    tmp0+=z1+z3;  tmp1+=z2+z4;  tmp2+=z2+z3;  tmp3+=z1+z4;

    /* combining the two parts */

    dataptr[0]=(DCTELEM) DESCALE(tmp10+tmp3, CONST_BITS-PASS1_BITS);
    dataptr[7]=(DCTELEM) DESCALE(tmp10-tmp3, CONST_BITS-PASS1_BITS);
    dataptr[1]=(DCTELEM) DESCALE(tmp11+tmp2, CONST_BITS-PASS1_BITS);
    dataptr[6]=(DCTELEM) DESCALE(tmp11-tmp2, CONST_BITS-PASS1_BITS);
    dataptr[2]=(DCTELEM) DESCALE(tmp12+tmp1, CONST_BITS-PASS1_BITS);
    dataptr[5]=(DCTELEM) DESCALE(tmp12-tmp1, CONST_BITS-PASS1_BITS);
    dataptr[3]=(DCTELEM) DESCALE(tmp13+tmp0, CONST_BITS-PASS1_BITS);
    dataptr[4]=(DCTELEM) DESCALE(tmp13-tmp0, CONST_BITS-PASS1_BITS);

    dataptr+=DCTSIZE;  /* advance pointer to next row */
  }
```

```
  /* Second Part: columns. */

  dataptr=data;
  for (rowctr=DCTSIZE-1; rowctr >= 0; rowctr--) {
    /* Even Part */

    z2=(int) dataptr[DCTSIZE*2];  z3=(int) dataptr[DCTSIZE*6];

    z1=z2+z3*FIX(0.541196100);
    tmp2=z1+z3*-FIX(1.847759065);  tmp3=z1+z2*FIX(0.765366865);

    tmp0=((int) dataptr[DCTSIZE*0]+(int) dataptr[DCTSIZE*4]) << CONST_BITS;
    tmp1=((int) dataptr[DCTSIZE*0]-(int) dataptr[DCTSIZE*4]) << CONST_BITS;

    tmp10=tmp0+tmp3;  tmp13=tmp0-tmp3;  tmp11=tmp1+tmp2;  tmp12=tmp1-tmp2;

    /* Odd part */

    tmp0=(int) dataptr[DCTSIZE*7];  tmp1=(int) dataptr[DCTSIZE*5];
    tmp2=(int) dataptr[DCTSIZE*3];  tmp3=(int) dataptr[DCTSIZE*1];

    z1=tmp0+tmp3;  z2=tmp1+tmp2;  z3=tmp0+tmp2;  z4=tmp1+tmp3;
    z5=z3+z4*FIX(1.175875602);

    tmp0=tmp0*FIX(0.298631336);  tmp1=tmp1*FIX(2.053119869);
    tmp2=tmp2*FIX(3.072711026);  tmp3=tmp3*FIX(1.501321110);
    z1=z1*-FIX(0.899976223);     z2=z2*-FIX(2.562915447);
    z3=z3*-FIX(1.961570560);     z4=z4*-FIX(0.390180644);

    z3+=z5;  z4+=z5;

    tmp0+=z1+z3;  tmp1+=z2+z4;  tmp2+=z2+z3;  tmp3+=z1+z4;

    dataptr[DCTSIZE*0]=(DCTELEM) DESCALE(tmp10+tmp3, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*7]=(DCTELEM) DESCALE(tmp10-tmp3, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*1]=(DCTELEM) DESCALE(tmp11+tmp2, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*6]=(DCTELEM) DESCALE(tmp11-tmp2, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*2]=(DCTELEM) DESCALE(tmp12+tmp1, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*5]=(DCTELEM) DESCALE(tmp12-tmp1, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*3]=(DCTELEM) DESCALE(tmp13+tmp0, CONST_BITS+PASS1_BITS+3);
    dataptr[DCTSIZE*4]=(DCTELEM) DESCALE(tmp13-tmp0, CONST_BITS+PASS1_BITS+3);

    dataptr++;   /* advance pointer to next column */
  }
}
```

9.2.2 Vectorization

Because there are no dependencies between different iterations of the loops, the code can be vectorized easily. We replace the first `for`–loop by vector code over the columns, and we replace the second `for`–loop by vector code over the rows, respectively. Accessing the matrix column wise requires to address memory with constant stride. We assume that at the beginning of the function the matrix is stored in *column wise* order. At the end of the first part, the matrix stored transposed. Thus, we need only once a constant stride access to memory.

We will not give a vectorized version of the code, because it is quite obvious how to obtain such a code: just replace the appropriate scalar variables with vector variables.

9.2.3 Dataflow Graphs

The three figures 4, 5 and 6 show the dataflow graphs of the three parts of one of the `for`–loops. The graphs for the other loop are identical. We have drawn the `shift`–operation as a unary operation, because the shift distance appears as a constant, which will be set only once. Parts of the constants which are used in the multiplications can be hold in registers and need not to be reloaded for the

second loop. The constants are loaded as immediate values coded in the instruction word. Thus, they do not interfer with vector transfers to or from memory. (Later, we will use subscribed numbers to distinguish the nodes of the three parts).

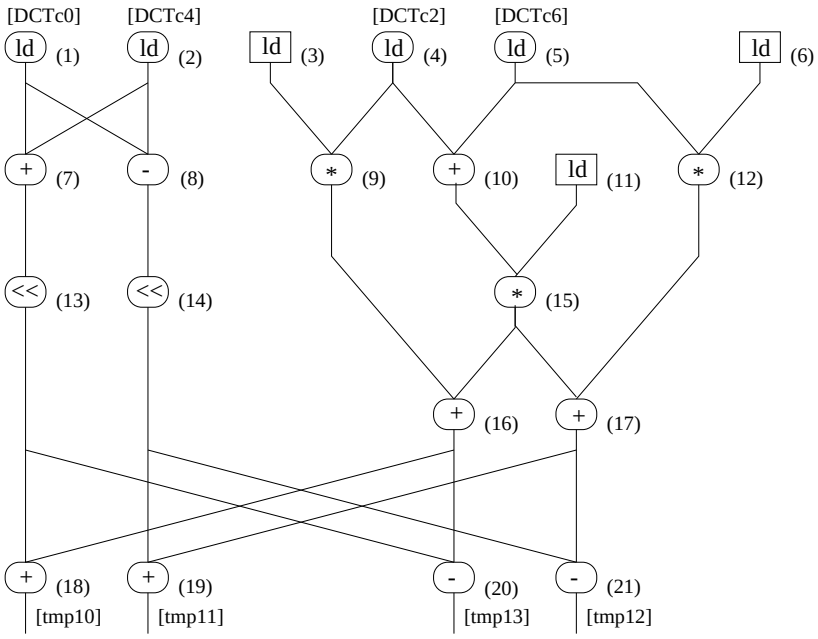

Figure 4: Dataflow graph for the even part

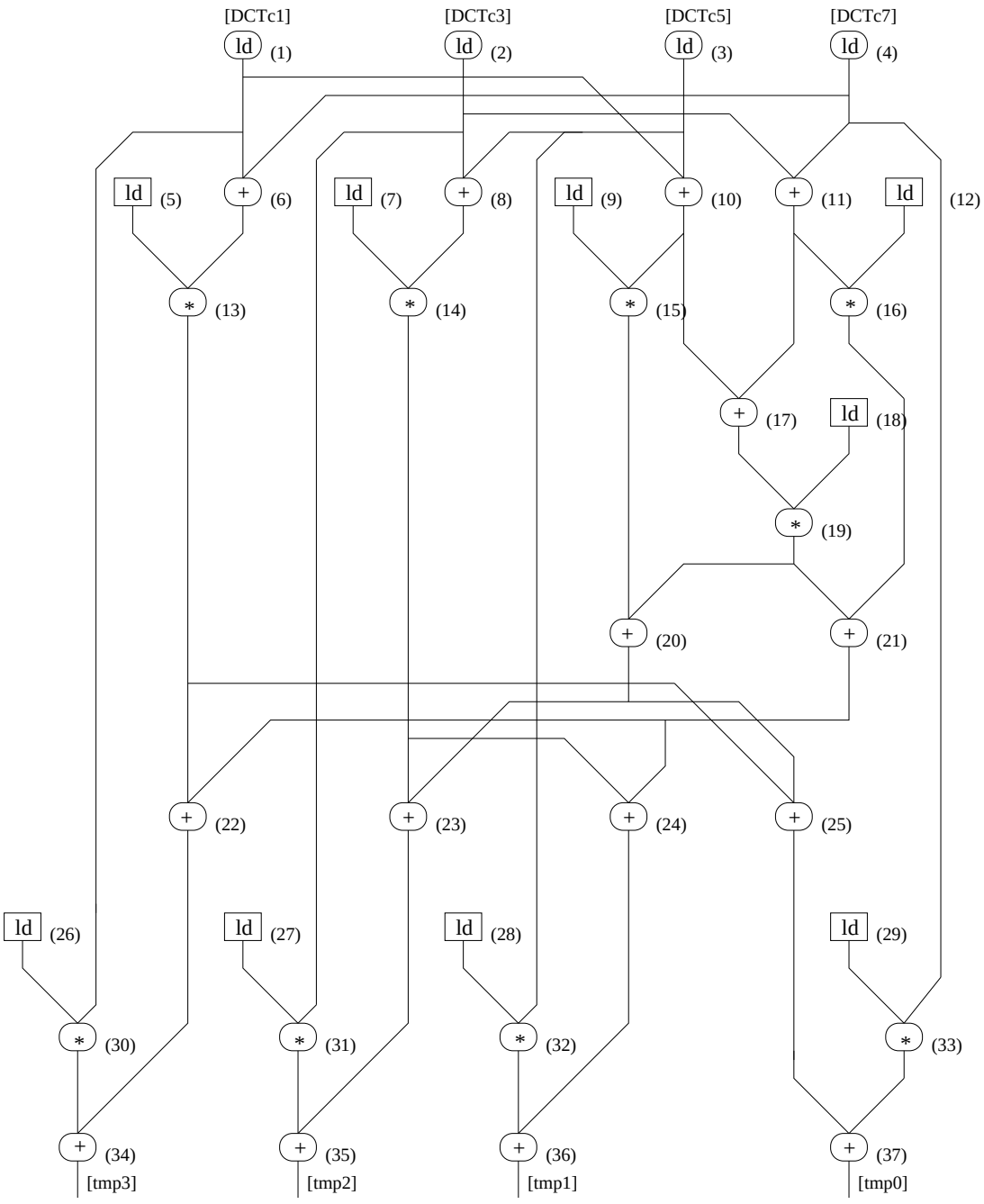

Figure 5: Dataflow graph for the odd part

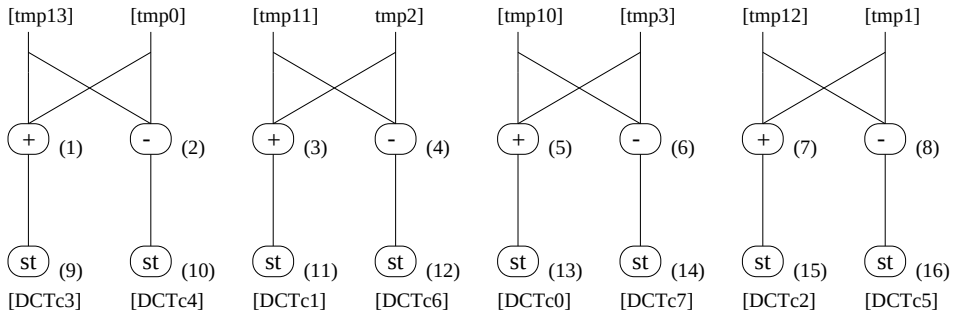

Figure 6: Dataflow graph for the combining of the even and odd part

9.2.4 Lower and Upper Bounds of the Run Time

The whole matrix must be loaded once in row wise and once in column wise order. Because we transpose the matrix in the program, storing can be done always row wise. Thus the lower bound due to memory access is 88 *cc*. If the vector length is increased to 16 or 32, the lower bound increases by a factor of 4 or 16 to 352 or 1408 *cc*. (Note that the number of function calls to `j_rev_dct()` will be reduced by the same factor).

For vector lengths of 16 or 32 we have to perform the program on larger matrices, which means that the number of operations and instructions is increasing by a factor of 2 or 4.

There are 32 vector load or store operations. Each of them is implemented with two instructions. There are 92 arithmetical operations, each counts one instruction. Twelve scalar fixed point values are used, which require 12 *cc* to be loaded. Thus the lower bound due to instructions is 168 *cc* and 336 or 672 *cc* for longer vectors, respectively.

There are less multiplications than other operations. All functional units can work in parallel. Thus, the lower bound due to availability of the functional units is equal to the total number of arithmetical operations divided by the number of functional units and multiplied by the time to deal with one vector. So we have 46, 184 and 736 *cc* for vector lengths 8, 16 and 32, respectively.

One longest path in the dataflow graph is given by the nodes:

$$(3)_2, (10)_2, (17)_2, (19)_2, (21)_2, (24)_2, (36)_2, (7)_3, (15)_3$$

which must be considered two times, where once a strided store is performed in node $(15)_3$. The number of cycles to go through that path is 57, 132 and 336 *cc* for vector lengths 8, 16 and 32, respectively. We have assumed that the basic blocks while dealing with larger vectors are not merged.

Hence, the program is for vector length 8 instruction issue limited—the lower bound is 168 *cc*—and for vector lengths 16 or 32 it is memory limited—with lower bound 352 or 1408. For vector length 16 the upper bound due to memory access and instruction issue are very close together, so that seems to be the most challenging case to try to optimize.

The upper bound for vector lengths 8 is three times 168 plus the time for the strided memory accesses, which sums to 560 *cc*. For vector length 16 we have three times 336 plus the time for the strided memory accesses, thus 1248 *cc*, and for vector length 32 we have four times 672 plus the time for strided memory accesses, thus 3680 *cc*.

The bounds are summarized in table 3.

9.2.5 Register allocation

Register allocation for the program sequence is straight forward. The first parts can be implemented with six vector registers and three scalar registers. The second part needs ten vector registers and

			vector length		
			8	16	32
lower bounds	memory	u_m	88	352	1408
	instructions	u_i	168	336	672
	operations	u_a	46	184	736
	dependencies	u_l	57	132	336
	maximal	u	168	352	1408
upper bound		v	560	1248	3680
minimal efficiency		η_{min}	0.30	0.28	0.38
possible improvement		q_{max}	3.33	3.57	2.63

Table 3: Lower and upper bounds for the run time of `j_rev_dct()`

nine scalar registers. The third part needs, besides the eight registers, which hold intermediate results of the previous two parts, only one additional vector register. We use only `t0` to `t8` as scalar registers, thus only six of the scalar values can be hold. Register reloading with constants does not increase the run time, because it can be hidden during the load of vectors with constant stride addressing.

9.2.6 Assembly program

We have implemented four versions of `j_rev_dct()`. Two versions handle the three parts in each loop of the program separately; two versions merge the parts, so that idle times can be avoided. The other difference is made according to the vector length (8 and 16 elements). The three parts could have been programmed using one dataflow graph, but the resulting graph is too large to be viewed entirely. Probably, one could run out of register, while trying to find an optimal schedule, and spilling of registers would become an other issue to be considered to.

Merging means that the loads and stores needed at the beginning of the second and third part are scheduled at the end of the previous part, so idle cycles can be avoided.

A fifth version uses the best routine for vector length 16 of the above to measure the run time for vector length 32. No special scheduling has been done for that case.

The comments in the assembly programs (Appendix) are refering to the appropriate nodes in the dataflow graphs.

9.2.7 Run Time

In table 4 the number of cycles for encached versions of the program and the efficiencies of the optimizations are listed. The column *achieved improvement* shows what improvement compared to a naive implementation has been achieved. They do not contain the initial settings of the vector length register or other constants, such as shift distance of stride register (see first lines of the assembly code).

The version with vector length 8 and merged parts shows only 28 idle cycles in the simulation. Those 28 cycles occur during the load operations with constant stride which in total take 64 cycles of the memory pipe. To hide more cycles than we have done, seems to be at least difficult.

In the version with vector length 16 and merged parts, there are 18 additional idle cycles which mostly are due to consecutive multiply instructions. The code has been optimized for vector length eight making the instruction issue the main bottleneck. Maybe it is possible to rearrange the instructions in a way that those cycles can be filled with other arithmetical operations.

vlr	merge	cycles	efficiency	improvement achieved	improvement possible
8	no	213	0.79	2.64	3.33
	yes	202	0.83	2.78	3.33
16	no	578	0.61	2.18	3.57
	yes	548	0.64	2.29	3.57
32	yes	1988	0.71	1.86	2.63

Table 4: Run time in *cc* and efficiency of `j_rev_dct()` for different versions

The best versions lead to the run times shown in table 5 on a 160 × 128 frame of the test film. We have added the times of a Sparc2 workstation as a reference. The run times are based on a 40 MHz chip of T0. The version with vector length 8 can be used more or less directly in the original version of the program. For the vector lengths 16 and 32 the code should be modified in such a way that four respectively 16 8 × 8–matrices can be handled simultaneously.

	Sparc2	Spert		
		8	16	32
iDCT	21.07 *ms*	2.08 *ms*	1.41 *ms*	1.27 *ms*

Table 5: Run time of TO for `j_rev_dct()` compared to measured run time on Sparc 2

10 Conclusion

We can summarize the following:

- It is not worth while to optimize non–encached code, i. e. if it is likely that the instructions cannot be held in the instruction cache, a simple code generator performing straight forward list scheduling would be sufficient.
- The introduced lower and upper bounds give some information about the performance of T0 on a program sequence and about the improvement one can get through code optimization.
- The method how to schedule instructions and how to allocate registers by hand leads to good results at least for the examined basis blocks.
- The bandwidth to memory—especially for strided and indexed access—seems to be the main bottleneck for the performance of T0.

For the MPEG–functions we have got the performance of T0 through the simulator listed in table 6.

References

[Le 91] Didier Le Gall. MPEG: a video compression standard for multimedia applications. *Communications of the ACM*, 34(4):46–58, April 1991.

[PSR93] Keton Patel, Brian C. Smith, and Lawrence A. Rowe. Performance of a software MPEG video decoder. In *Proceedings of the ACM Multimedia Conference*, 1993.

	η_{min}	η	q	μs	speed up
Dithering	0.33	1.00	3.00	0.61	52.4
Reverse DCT	0.38	0.71	1.86	1.27	16.6

Table 6: Performance of T0 (best cases) and comparison to Spark 2

A Example of Assembly Program

```
#include "tor/tordef.h"

.data
        .align  4
DCT:    .space 128 # DCT input matrix: 64 16-bit values

 # the matrix is given in row wise order
 # the matrix is returned in column wise order
 # thus only once a memory operation with stride must be performed

 # the rows are named DCTr0 to DCTr7
 # the columns        DCTc0 to DCTc7 repectively

 # the routine should be modified to receive a pointer to the matrix,
 # but for now I just use a fixed memory location

 # a loop is added in order to encache the code

.text
.align 4
        .globl  idct
        .ent    idct
idct:

        li      v1, 16 # stride 8*sizeof(hword)
li v0, 8 # used for shift distance and vlr
ctvu v1, VLR # vector length set
li      t9, 2 # loop counter
loop:
la      a0, DCT # normally as parameter

        # First part: rows
#   even columns
add a0, 96 # DCTc6
  lh.v vv1, a0 # ( 4)_1
li t0, 1 # ( 3)_1
sub a0, 64 # DCTc2
lh.v vv2, a0 # ( 5)_1
hmul.vs vv3, vv1, t0 # ( 8)_1
sub a0, 32 # DCTc0
lh.v vv4, a0 # ( 1)_1
add.vv vv1, vv1, vv2 # (10)_1
add     a0, 64 # DCTc4
lh.v vv5, a0 # ( 2)_1
```

```
li t1, 2 # ( 6)_1
li t2, 3 # (11)_1
hmul.vs vv2, vv2, t1 # (12)_1
sub.vv vv6, vv4, vv5 # ( 8)_1
hmul.vs vv1, vv1, t2 # (15)_1
add.vv vv4, vv4, vv5 # ( 7)_1
sllv.vs vv5, vv6, v0 # (14)_1
add.vv vv2, vv1, vv2 # (17)_1
add.vv vv1, vv1, vv3 # (16)_1
sllv.vs vv3, vv4, v0 # (13)_1
sub.vv vv4, vv5, vv2 # (21)_1
add.vv vv2, vv5, vv2 # (19)_1
sub.vv vv5, vv3, vv1 # (20)_1
add.vv vv1, vv3, vv1 # (18)_1

#   Odd columns
sub a0, 16 # DCTc3
lh.v vv3, a0 # ( 2)_2
li t3, 4 # (28)_2
li t4, 5 # (29)_2
li t5, 6 # (12)_2
li t6, 7 # ( 7)_2
li t7, 8 # ( 9)_2
li t8, 9 # (18)_2
add a0, 64   # DCTc7
lh.v vv6, a0 # ( 4)_2
li t0, 10 # (26)_2
li t1, 11 # ( 5)_2
li t2, 12 # (27)_2
sub a0, 32 # DCTc5
lh.v vv7, a0 # ( 3)_2
sub a0, 64 # DCTc1
lh.v vv8, a0 # ( 1)_2
hmul.vs vv9, vv7, t3 # (32)_2
add.vv vv10, vv3, vv6 # (11)_2
hmul.vs vv11, vv6, t4 # (33)_2
add.vv vv12, vv3, vv7 # ( 8)_2
add.vv vv7, vv7, vv8 # (10)_2
hmul.vs vv13, vv10, t5 # (16)_2
hmul.vs vv12, vv12, t6 # (14)_2
add.vv vv10, vv10, vv7 # (17)_2
hmul.vs vv7, vv7, t7 # (15)_2
add.vv vv6, vv6, vv8 # ( 6)_2
hmul.vs vv10, vv10, t8 # (19)_2
hmul.vs vv8, vv8, t0 # (30)_2
hmul.vs vv6, vv6, t1 # (13)_2
add.vv vv7, vv7, vv10 # (20)_2
add.vv vv10, vv10, vv13# (21)_2
hmul.vs vv3, vv3, t2 # (31)_2
add.vv vv13, vv6, vv7 # (25)_2
add.vv vv14, vv10, vv12# (24)_2
add.vv vv7, vv7, vv12 # (23)_2
add.vv vv6, vv6, vv10 # (22)_2
add.vv vv10, vv11, vv13# (37)_2
add.vv vv3, vv3, vv7 # (35)_2
```

```
add.vv vv6, vv6, vv8 # (34)_2
add.vv vv7, vv9, vv14 # (36)_2

# Combining
add.vv vv8, vv5, vv10 # ( 1)_3
sub.vv vv5, vv5, vv10 # ( 2)_3
add a0, 32 # DCTc3
sh.v vv8, a0 # ( 9)_3
add a0, 16 # DCTc4
add.vv vv8, vv2, vv3 # ( 3)_3
sh.v vv5, a0 # (10)_3
sub.vv vv2, vv2, vv3 # ( 4)_3
sub a0, 48 # DCTc1
sh.v vv8, a0 # (11)_3
add a0, 80 # DCTc6
add.vv vv8, vv1, vv6 # ( 5)_3
sh.v vv2, a0 # (12)_3
sub.vv vv1, vv1, vv6 # ( 6)_3
sub a0, 96 # DCTc0
sh.v vv8, a0 # (13)_3
add a0, 112 # DCTc7
add.vv vv8, vv4, vv7 # ( 7)_3
sh.v vv1, a0 # (14)_3
sub.vv vv2, vv4, vv7 # ( 8)_3
sub a0, 80 # DCTc2
sh.v vv8, a0 # (15)_3
add a0, 48 # DCTc5
sh.v vv2, a0 # (16)_3

# Second part: columns
#   even rows
sub a0, 68 # DCTr6
  lhst.v vv1, a0, v1 # ( 4)_1
li t0, 1 # ( 3)_1
sub a0, 8 # DCTr2
lhst.v vv2, a0, v1 # ( 5)_1
hmul.vs vv3, vv1, t0 # ( 8)_1
sub a0, 4 # DCTr0
lhst.v vv4, a0, v1 # ( 1)_1
add.vv vv1, vv1, vv2 # (10)_1
add     a0, 8 # DCTr4
lhst.v vv5, a0, v1 # ( 2)_1
li t1, 2 # ( 6)_1
li t2, 3 # (11)_1
hmul.vs vv2, vv2, t1 # (12)_1
sub a0, 2 # DCTr3
lhst.v vv15, a0, v1 # ( 2)_2
sub.vv vv6, vv4, vv5 # ( 8)_1
hmul.vs vv1, vv1, t2 # (15)_1
add.vv vv4, vv4, vv5 # ( 7)_1
sllv.vs vv5, vv6, v0 # (14)_1
add.vv vv2, vv1, vv2 # (17)_1
add.vv vv1, vv1, vv3 # (16)_1
sllv.vs vv3, vv4, v0 # (13)_1
add a0, 8   # DCTr7
```

```
lhst.v vv6, a0, v1 # ( 4)_2
li t0, 10 # (26)_2
li t1, 11 # ( 5)_2
li t2, 12 # (27)_2
sub a0, 4 # DCTr5
sub.vv vv4, vv5, vv2 # (21)_1
add.vv vv2, vv5, vv2 # (19)_1
sub.vv vv5, vv3, vv1 # (20)_1
add.vv vv1, vv3, vv1 # (18)_1

#   Odd columns
lhst.v vv7, a0, v1 # ( 3)_2
sub a0, 8 # DCTr1
lhst.v vv8, a0, v1 # ( 1)_2
hmul.vs vv9, vv7, t3 # (32)_2
add.vv vv10, vv15, vv6 # (11)_2
hmul.vs vv11, vv6, t4 # (33)_2
add.vv vv12, vv15, vv7 # ( 8)_2
add.vv vv7, vv7, vv8 # (10)_2
hmul.vs vv13, vv10, t5 # (16)_2
hmul.vs vv12, vv12, t6 # (14)_2
add.vv vv10, vv10, vv7 # (17)_2
hmul.vs vv7, vv7, t7 # (15)_2
add.vv vv6, vv6, vv8 # ( 6)_2
hmul.vs vv10, vv10, t8 # (19)_2
hmul.vs vv8, vv8, t0 # (30)_2
hmul.vs vv6, vv6, t1 # (13)_2
add.vv vv7, vv7, vv10 # (20)_2
add.vv vv10, vv10, vv13# (21)_2
hmul.vs vv3, vv15, t2 # (31)_2
add.vv vv13, vv6, vv7 # (25)_2
add.vv vv14, vv10, vv12# (24)_2
add.vv vv7, vv7, vv12 # (23)_2
add.vv vv6, vv6, vv10 # (22)_2
add.vv vv10, vv11, vv13# (37)_2
add.vv vv3, vv3, vv7 # (35)_2
add.vv vv6, vv6, vv8 # (34)_2
add.vv vv7, vv9, vv14 # (36)_2

# Combining
add.vv vv8, vv5, vv10 # ( 1)_3
sub.vv vv5, vv5, vv10 # ( 2)_3
add a0, 4 # DCTr3
sh.v vv8, a0 # ( 9)_3
add a0, 2 # DCTr4
add.vv vv8, vv2, vv3 # ( 3)_3
sh.v vv5, a0 # (10)_3
sub.vv vv2, vv2, vv3 # ( 4)_3
sub a0, 6 # DCTr1
sh.v vv8, a0 # (11)_3
add a0, 10 # DCTr6
add.vv vv8, vv1, vv6 # ( 5)_3
sh.v vv2, a0 # (12)_3
sub.vv vv1, vv1, vv6 # ( 6)_3
sub a0, 12 # DCTr0
```

```
sh.v vv8, a0 # (13)_3
add a0, 14 # DCTr7
add.vv vv8, vv4, vv7 # ( 7)_3
sh.v vv1, a0 # (14)_3
sub.vv vv2, vv4, vv7 # ( 8)_3
sub a0, 10 # DCTr2
sh.v vv8, a0 # (15)_3
add a0, 6 # DCTr5
sh.v vv2, a0 # (16)_3

sub t9, 1
bgtz t9, loop
nop
j       ra

.end    idct
```