Hasenjaeger's Register Machine with Wang Instructions

Rainer Glaschick, Paderborn, Germany
2013-06-24

1. Introduction

In 1984, Gisbert Hasenjaeger published (in [Hasenjaeger1984]) the state table for a machine with only two states, designed to use non-writing turing tapes as registers. The description is very concise, as it served only to illustrate his main subject, the use of Jones-Matiyasevich masking.

This machine is discussed here in more detail. The reason is primarily to be able to identify if any device of his legacy is an attempt to build such a machine. In particular, there is an object with three counters, (see picture in the Appendix), that might have a relation to the publication, and could have been used together with the *Old Wang*. While writing, however, more notes about materialization of a Turing machine with non-writing tapes piled up, a subject which, to my knowledge, did not receive much attention yet.

As a side effect in thinking about materialization of such a machine, a smaller machine (in section 3.4, Variable coding with three instructions) has been found with still only two states, but a smaller TM index, only topped by Neary and Wood's rule-110 based machine published in 2009.

Register machines (also called counter machines or Minsky machines) were first published by Shepherdson and Sturgis [ShepherdsonSturgis1963] and got broader attention by Minsky's book [Minsky1967]. According to Hasenjaeger [Hasenjaeger1987], Rödding independently developed similar ideas, but refrained from publication when they saw Minsky's article [Minsky1961] using non-writing Turing tapes to hold integer values just by using the position from the single (end) mark.

Rödding later described his achievements in [Roedding1972], refering to Shepherdson and Sturgis as only source. A working demonstrator for register machines was built by Carsten Carstensen in 1975, see [Carstensen1975]. Register machines have be further studied and refined by Elmar Cohors-Fresenborg, see his German book [CohorsFresenborg1977].

Hasenjaeger (in [Hasenjaeger1987]) coined the term *materialization* for building physical demonstrator machines, which will also be used in this text. Included are also remarks on building such machines in real as demonstrator objects for educational purposes.

Complexity issues are only hinted in the last section.

2. Register machines revisited

In 1987, Hasenjaeger wrote in [Hasenjaeger1987] that the problem of crafting physical Turing tapes inspired Dieter Rödding and himself to try to use non-writing Turing tapes instead of non-erasable Turing tapes, and that they proved that such a machine is as powerful as any turing machine.

Non-writing Turing tapes have a single mark only, and are used merely as counters for integer numbers, where the number of fields between the current field and the single marked field is representing the integer. The machine thus could increment and decrement a number, and sense whether the number is zero (tape is on the mark). Personally I would prefer *immutable* or *read-only* over the clumsy *not-writing* attribute, but *non-writing* is established enough to be kept.

Although already Minsky has proven ([Minsky1967], Theorem 14.2-1) that one register is sufficient, using the encoding of a set of numbers as exponents of prime numbers, this slows down the operation by orders of magnitude. Note that the counter tapes have the mark at the fixed end; they are not moved backwards over the mark, representing non-negative integer numbers only. Any move beyond that limit is normally suppressed, if not resulting in a machine stop.

Besides the registers, there is a state machine, often written as a squence of instructions or programme, to conditionally move the tapes.

Often the number of registers is not limited a-priori, and number of counters actually used depends on the programme. Minsky proved that two counters are sufficient, and Cohors-Fresenborg three with structured loops. For practical demonstrations, at least four are useful, as Carstensen used in his machine.

2.1. Minsky's instructions

Minsky in his book [Minsky1967] uses in 11.1 four primitive instructions:

- zero a register
- increment a register
- jump if zero, else decrement
- halt

A fifth instruction, the unconditional goto, is used as an abbreviation to setting register w zero followed by a conditional jump, which jumps because w is zero. The easiest way to achieve this is to set aside a never changed register, e.g. register zero, and only use it for unconditional jumping.

Zeroing a register is easy to accomplish by a small loop jumping around an unconditional backward jump, but clearly costs time[1]. . Halt may be dispensible or realized, as Hasenjaeger called it, by a *dynamic stop*, i.e. a tight loop jumping to itself.

In section 14.1, he uses a reduced model:

- a': Add unity to a register a
- $a^{-}(n)$: If register a is zero, jump to destination n; otherwise, decrement the register (and do not jump)

He then again argues that an unconditional goto, written as $go(n)$, could always be done by $w^{-}(-1)$, provided that there is a register w which is already zero, and clearing a register, written as a^{0}, via $a^{-}(+1)\ go(-1)$. In the following diagrams, the need for a third register that stays to zero is not mentioned.

For Theorem 14.1-1, proving that two registers are sufficient, however, he uses yet another instruction set, in that

> *The machine uses only the operations $'$ and $-$, assuming that the successor instruction contains the "go" information for the next instruction.*

As long as the unconditional go is preceeded by an increment, this works directly; otherwise, a decrement has to inserted at the jump target[2]. .

The use of the next instruction address in every instruction is remarkable, as the same was used in some early delay memory computers, such as Turing's ACE.

2.2. Roedding's (structured) instructions

Using Minsky's instructions might lead to arbitrary complex programs, although his examples follow a rather structured approach. Having the conditional jump *on zero*, his loops are in general *while loops*, where the loop is skipped on entry, and repeated with an unconditional jump at the end. Conditional jumps allow to leave a loop before repetition, and this is used fairly often.

Dieter Rödding (in [Roedding1972]) uses two elemenatary operations, namely increment and decrement (limited to zero) of a certain register, and combines them together with an iteration that repeats any composite machine while a register is not zero. This is denoted by putting the composite machine into parenthesis and indexing the closing parenthesis with the register number. Note that due to the concept of a *composite machine*, the machine inside the parenthesis is not executed if the register that indexed the closing parenthesis is zero. This textual machine description has its equivalence in *Structured programming*, using only *while*-loops.

To erase register `i`, the notation is thus written in Roedding's notation:

$$(S_i)_i$$

Adding register 2 to register 1 (clerаring register 2) reads:

$$(A_1 S_2)_2$$

To materialize such a machine (see [Carstensen1975]), an unconditional jump goes to the closing parenthesis, which is a conditional jump back behind the opening parenthesis if the register denoted by the index is not zero. Carstensen indexes the opening parenthesis instead, so that it was clear that the loop is executed only if the given register is not zero, but could have also used Roedding's notation directly with a slightly more complex hardware.

By prefixing the indices to the symbol, and using `+` and `-` instead of `A` and `S`, the programmes can be written without indices[3], the above examples become:

```
( 1- 1)         clear r1
( 1+ 2- 2)      add r1 to r2, clearing r1
```

which is used from now on, writing `r3` for *register 3* in the comments and treating the first letter and what follows as comment. Also, all registers that are not input parameters, must be zero upon start of the examples.

Producing the sum of `r1` and `r2` in `r3` becomes:

```
( 3+ 1- 1)      move r1 to r3
( 3+ 2- 2)      add r2 to r3
```

Note that the decrement of the loop register can (always?) be moved to the end, so we use `3-)` as an abbreviation for `3- 3)` furtheron. Auxiliary registers that are zero on entry and left zero are denoted using register numbers `9, 8, ...` and register names `t9, t8, ....`

While we will prefix the register numbers directly to the operations, they could be interpreted as instructions on its own, setting the *current register*, and allowing `2 - -` instead of `2- 2-`. Here this option is not further used, because seldom applicable, but it is a central concept in later sections.

In the following examples, **keep in mind that the loops are *while loops***.

To have the sum of `r1` and `r2` in `r3`, but save the values of `r1` and `r2` gives:

```
( 3+ 9+ 1-)     add r1 to r3 and copy to auxiliary
( 1+ 9-)        restore r1
( 3+ 9+ 2-)     add r2 to r3 and copy to auxiliary
( 2+ 4-)        restore r2
```

To subtract `r2` from `r1` clearing `r2` gives:

```
( 1- 2-)
```

Note that, if the result would be negative, it is zero instead, as the use of negative numbers is not directly supported.

To multiply `r1` by `2` reads:

```
( 9+ 8+ 1-)     move r1 to t9 and t8
( 8+ 9-)        add t9 to t8
( 1+ 8-)        move result from t8 to r1
```

Multiplication of `r1` with `r2` giving `r3` repeatedly adds `r2` to `r3`, both operands saved:

```
(
 ( 3+ 9+ 2-)    add r2 to r3 and save in t9
 ( 2+ 9-)       restore r2
1-)             while r1 is not zero
```

Dividing `r1` by `2` may be attempted by:

```
( 9+ 1- 1-)     subtract twice in loop
( 1+ 9-)        result back to r1
```

This is fine for even numbers, but one more for odd numbers, so we decrement before the loop[4]:

```
( 9+ 1-)        to t9
9-              in case r1 is odd
( 1+ 9- 9-)     decrement twice for each round
```

For calculating the remainder of a division, e.g. by 2, the solution given by Cohors-Fresenborg in exercise 6.11 divides, multiplies and uses the difference, taking about `3n+4n/2+2n/2+2n=8n` steps and needs 3 temporary registers:

```
( 9+ 8+ 1-)     to x9 and x8
8-              compensate odd input
( 1+ 7+ 8- 8-)  divide by 2 to r1 and x7
( 1+ 7-)        add gives times 2
( 9- 1-)        difference is remainder
( 1+ 9-)        result to r1
```

Dividing both, n and $n + 1$ and subtracting the difference during the second division takes about `3n+3n+3n=9n` steps and is slower, but shorter and needs only 2 temporary registers:

```
( 9+ 8+ 1-)     to x9 and x8
( 1+ 9- 9-)     divide x9 by 2, one off if odd
8-
( 1- 8- 8-)     divide x8 by 2, subtract from r1
```

Calculating quotient and remainder together uses about `3n+5n=8n` steps and not slower, but shorter:

```
( 9+ 2+ 1-)     to t9 and r2
9-              truncate loop early if odd
( 1+ 2- 2- 9- 9-)
```

If the remainder is not needed, a single `2-` will clear it.

2.3. Extensions: loop break and conditional

With arbitray goto operations, the solution is much easier, as the loop can be terminated early. Using the additional operator `r>` to leave the loop, if register `r` is zero, i.e. skip beyond its end, the division becomes:

```
( 1- 1>  9+ 1-)
( 1+ 9-)
```

and the remainder of the division by 2:

```
( 9+ 1- 1> 9- 1-)       leave loop if r1 was odd
( 1+ 9-)
```

Alternatives would be *if* conditionals, where only the comparison to zero would be needed. Let `1[` make the following text execute only if register `1` is not zero, until the closing `]` is found, then the division would read:

```
( 1- 1[ 9+ ] 1-)
```

And the remainder calculated by:

```
( 1- 1[ 9+ ] 9-   1-)
```

Note that even if the brackets are replaced by parenthesis, the formula is still properly nested.

Note that a `leave` is different from a condition embracing the rest of the loop, as it terminates the loop without clearing the loop counter.

As far as I can see, branches, i.e. case decisions as are often used in function definitions, see Collatz series , are modeled in the theory of recursive functions by calculating the conditions as zero or one, evaluating all possible function values, and multiplying them with the zero or one values, adding up the sum. While this works, it is, from a practical point of view, awfully inefficient.

2.4. Encoding a Turing tape contents by an integer number

Encoding a Turing tape including its position in a single integer number is done by Minsky [Minsky1961] representing it by the number $2^x \cdot 3^{2^k}$, where k is the tape contents as binary number, and x the position of the current square. In order to operate the machine, besides an auxiliary tape, rather extensive and expensive calculations are used.

An alternate version (published where ???) uses two counter tapes, one containing the left, the other one the right hand part seen from the read-write head of a binary turing tape. To move the simulated tape left, we have to divide the value representing the left part by two, multiply the other tape by two, and add one to the second tape if the new symbol there should be a one. This obviously requires a third tape too, to receive the halved and doubled values, but the arithmentic is much moderate.

3. Hasenjaeger's proposed machine

In [Hasenjaeger1984], a state table is given for a Turing Machine emulating a register machine with three registers, with three (non-writing) counting tapes, R_0, R_1 and R_2, an auxiliary counting tape J to count jump destinations, and a programme tape P. The original state table is given in the Appendix, but first the scene will be stuffed a bit.

3.1. Preparations

The necessary instructions are:

- advance forward (increment) a tape,
- decrement it (advance back),
- jump to a different instruction if the tape is — or is not — at its beginning, i.e. is zero.

With three tapes, there are six instructions for increment and decrement, and three times the jump instructions with a variable target, depending on one of the tapes.

To denote a jump with certain distance, the jump instruction is just repeated the necessary number of times, as two or more jumps on the same condition in succession are not needed.

This gives 9 instructions, which requires 4 bits to encode in binary. Let us denote the tape with a index, then we have

e_i increment (enlarge) register i

d_i decrement register i if greater zero

$j_i(n)$ jump if register i is not zero

The number n used in the jump instruction denotes a relative distance, thus it will always be written with a sign (zero may be used as a substitute for halt).

Clearing register 1 is done by

$$d_1 j_1(-1)$$

If d_1 was zero initially, the first decrement does nothing (possibly by inhibiting the move if the tape is marked), and the jump continues instead of jumping. However, the loops used here are of the *do loop* kind, which test the jump condition at the end of the loop, and are thus vulnerable on border cases. This problem will mostly be neglected and only shown occasionally for illustration purpose.

Adding register 1 to register 2 is obtained by

$$j_1(+3) e_2 d_1 j_1(-2)$$

To reduce the number of instructions, Hasenjaeger very early had the idea to have a current tape, and to rotate through the tapes cyclically, or one may say, to have a sliding window that opens to just one tape or register.

So this gives four instuctions (instead of nine) that can be coded in two binary digits:

```
c  11   Cycle, i.e. change tape window cyclically
d  10   Decrement current tape
e  01   Enlarge current Tape
f  00   iF, jump conditionally if the current tape is not zero
```

These are encoded very close Hooper's Wang instructions as in Appendix II of [Hooper1969].

So the coding on the programme tape is a sequence of bit pairs to be denoted here by the four characters above.

The (conditional) jump `f` uses the observation that two jumps in sequence are not needed, thus the distance can be indicated by a sustained sequence of letters `f`, where the count denotes the distance.

If the skip would be backwards and the jump applied if the register is not zero, clearing a register would be very short:

```
d f
```

To add the current register to the next one becomes:

```
c e c c d f f f f f
```

Note that the number of changes within a loop is always a multiple of the number of tapes, here 3.

Hasenjaeger's machine, however, does not jump backwards, but forward, and uses a cyclic tape to achieve backward jumps by long forward jumps. This is possible as the number of `f` letters gives the number of non-`f` letters to skip; as backwards loops are very common, the programme tape is filled mostly with letters `f`.

The machine state table is now simpler, as the programme tape is advanced once for each instruction, so the column for the action on `P` can be left out[5].

3.2. The state table rewritten

The machine table in Appendix is rewritten here, with the letters for the programme tape inscription instead of the binary encoding, and columns and goups rearranged. Blank positions in the original state table seem to denote any input, while here a dot is used; blank space for states means *same as above*. `S` denotes the state, `J` the jump counter tape, `R` the threefold result tape, and `P` the programme tape, small letters for the new values:

```
S PRJ   rj s    Remark
                Group 1: sequential instructions
0 c.0   #. .    cycle tape
  d10   -. .    decrement register
  e.0   +. .    increment register
  f.0   .. 1    start jumping
                Group 2: executing a jump
0 c.1   .- .    decrement J for c
  d.1   .- .    decrement J for d
  f.1   .. .    skip  f
                Group 3: do not jump as R is zero
1 c0.   .. 0    terminated by c
  d0.   .. 0    terminated by d
  f0.   .. .    ignore f
                Group 4: collect jump distance
1 c1.   .. 0    found c
  d1.   .. 0    found d
  f1.   .+ .    count number of 'f's
```

There are some state combinations missing, these are

```
S PRJ
0 d00           decrement while R is zero
1 e0.           terminated by e
1 e1.           found e
```

The first one is obvious; nothing has to be done for a decrement instruction on a zero R tape. The others suggest that a chain of `f` instructions is never followed by an `e` instruction. This is weird; on the contrary, a `d` instruction after a `f` instruction is useless: either the `f` did not jump because the current register is zero, then `d` is suppressed; or `f` did jump, then `d` is ignored; in any case, a `d` after an `f` is useless, unless being a jump target. (The sequence `fd` will be used as a halt later.) On the other hand, `f e` might be useful to set the just cleared register to 1.

Encountering a `f` always changes to state 1 without incrementing `J`, which is ok for group 4, as the terminating non-`f` is already skipped before we go back to state 0. In the case of the current register zero, one instruction is skipped always, even if no jump was intended. This may have been overlooked by Hasenjaeger; an easy solution is to skip `f` if `R=0` unconditionally.

This gives a modified table assumed to better cover the intentions:

```
S PRJ   rj s    Remark
                Group 1: sequential instructions
0 c.0   #. .    cycle tape
  d00   -. .    suppress decrement as register is zero
  d10   -. .    decrement register
  e.0   +. .    increment register
  f00   .. .    skip f as register is zero
  f10   .. 1    start jumping
                Group 2: executing a jump
0 c.1   .- .    decrement J for c
  d.1   .- .    decrement J for d
  e.1   .- .    decrement J for e
  f.1   .. .    skip  f
                Group 3: (moved to state 0)
                Group 4: collect jump distance
1 c1.   .. 0    found c, skipped
  d1.   .. 0    found d, skipped
  e1.   .. 0    found e, skipped
  f1.   .+ .    count number of 'f's
```

Regarding the TM index (see [Glaschick2012]), all $5 \cdot 2 \cdot 2 = 20$ input symbols are used, and the 6 output actions are `#. -. +. .- .+ ..`, thus the basic TM index is $2 \cdot \left(\sqrt{\frac{20 \cdot 6}{2}}\right) = 2 \cdot \sqrt{60} = 15.5$. For the cyclic programme tape, three instead of one output tape, and a missing stop, a penalty of 4 output actions is used, giving an estimated TM index of $2 \cdot \left(\sqrt{\left(20 \cdot \frac{6+4}{2}\right)}\right) = 2 \cdot \sqrt{110} = 20.0$, which is lower than Hasenjaeger's Mini-Wang with 24.0 and 26.5. Note that the benefit from non-writing tapes materializes in the small number of output actions. Supressing the decrement in the tape drive is possible, but does not change the TM index.

Note that splitting state 0 on `J` looks like state expansion using tape `J`, while a straigthforward solution would use a state on its own. This, is, however not the case, as `J` contains the skip number; moreover, a different state would either make advancing `P` optional, or duplicate actions in state 0, see later examples.

To implement the stop, where no tape is moved any longer, including the programme tape, either the advance of P would be included and optional, or a fake state h is used to stop the clock generator. In either case a version is required that changes state to control skipping, i.e. uses group 3 again, and duplicates group 1 actions.

Using fake state h as stop:

```
S PRJ   rj s    Remark
                Group 1: sequential instructions
0 c.0   #. .    cycle tape
  d00   -. .    suppress decrement as register is zero
  d10   -. .    decrement register
  e.0   +. .    increment register
  f.0   .. 1    start jumping
                Group 2: executing a jump
0 c.1   .- .    decrement J for c
  d.1   .- .    decrement J for d
  e.1   .- .    decrement J for e
  f.1   .. .    skip  f
                Group 3: do not jump as R is zero
1 c0.   #. 0    found c, execute and proceed with group 1
  d0.   .. h    found d, this is a stop
  e0.   +. 0    found e, execute and proceed with group 1
  f0.   .. .    skip any f
                Group 4: collect jump distance
1 c1.   .. 0    found c
  d1.   .. 0    found d
  e1.   .. 0    found e
  f1.   .+ .    count number of 'f's
```

Now there is an extra output symbol, thus the basic TM index is $2 \cdot \left(\sqrt{\frac{20 \cdot 7}{2}}\right) = 2 \cdot \sqrt{70} = 16.7$. The estimated TM index is the same 20.0, as one less penalty for the missing halt is compensated by one extra output symbol.

Using control of P:

```
S PRJ   prj s   Remark
                Group 1: sequential instructions
0 c.0   +#. .   cycle tape
  d00   +-. .   suppress decrement as register is zero
  d10   +-. .   decrement register
  e.0   ++. .   increment register
  f.0   +.. 1   start jumping
                Group 2: executing a jump
0 c.1   +.- .   decrement J for c
  d.1   +.- .   decrement J for d
  e.1   +.- .   decrement J for e
  f.1   +.. .   skip  f
                Group 3: do not jump as R is zero
```

```
1 c0.   +#. 0   found c, execute and proceed with group 1
  d0.   ... .   found d, this is a stop
  e0.   ++. 0   found e, execute and proceed with group 1
  f0.   +.. .   skip any f
                Group 4: collect jump distance
1 c1.   +.. 0   found c
  d1.   +.. 0   found d
  e1.   +.. 0   found e
  f1.   +.+ .   count number of 'f's
```

The TM indices remain the same, as the number ouf output actions is unchanged.

3.3. Integrating decrement with skip

As mentionend above, with proper ordering of the instructions, a loop jump is always preceeded by a decrement of that register. Thus, the loop could decrement, requiring only three instructions:

```
c  cycle tapes
e  increment
f  decrement if not zero and jump if result is not zero
```

However, as the division example shows, the decrement is sometimes needed outside a loop, which could be done (for 3 tapes) by

```
fffccc
```

If the current register is zero, the `f` letters are skipped, and the three `c` cycle back to the same register. If the current register is one, it will be decremented to zero, and go on as before. If the current register is greater than one, it will be decremented, and the skip forward activated, skipping the three cycle instructions. Thus, while using much more space for a single decrement, the number of instructions can be reduced and the state table made smaller.

Unfortunately, the use of `fd` for stop is no longer possible; so the smaller state table without stop is modified:

```
S PRJ   rj s    Remark
                Group 1: sequential instructions
0 c.0   #. .    cycle tape
  e.0   +. .    increment register
  f00   .. .    skip f as register is zero
  f10   -. 1    decrement and start jumping
                Group 2: executing a jump
0 c.1   .- .    decrement J for c
  e.1   .- .    decrement J for e
  f.1   .. .    skip  f
                Group 3: zero after decrement
1 c0.   #. 0    found c, execute and restart
  e0.   +. 0    found e, execute and restart
  f0.   .. 0    skip this f
```

```
                   Group 4: collect jump distance
        1 c1.   .. 0    found c, skipped
          e1.   .. 0    found e, skipped
          f1.   .+ .    count number of 'f's
```

All $3 \cdot 2 \cdot 2 = 12$ input symbols are used, the output symbols are still 6, thus the basic TM index is $2 \cdot \left(\sqrt{\frac{12 \cdot 6}{2}}\right) = 2 \cdot \sqrt{36} = 12.0$ instead of 15.5. Still a penalty of 4 is appropriate, so the estimated TM index is $2 \cdot \left(\sqrt{\frac{12 \cdot (6+4)}{2}}\right) = 2 \cdot \sqrt{60} = 15.5$ instead of 20.0. This solution has better TM indices than the optimized Mini-Wang with 14.7 and 15.9.

If a skip to itself is accepted as a dynamic stop, the estimated TM index would use a penalty of 3 instead of 4, giving $2 \cdot \left(\sqrt{\frac{12 \cdot (6+3)}{2}}\right) = 2 \cdot \sqrt{54} = 14.7$.

3.4. Variable coding with three instructions

Applying the variable length encoding of actions used in the Mini-Wang uses a binary programme tape and the following instructions:

```
        1     cycle registers
        01    increment current register
        00ⁿ1  decrement (if not already zero) and skip if not zero
```

The skip instruction skips the following n 1 marks on the tape, thus always positions behind an instruction. There is no halt; a skip to self may be used as dynamic stop.

A first unpolished version of a state table might be:

```
        S PRJ  rj s     Comments
        1 1.0  c. .     cycle instruction
          0.0  .. 2     incr or skip
          111  .- .     skip ones and count
          011  .. .     skip zeros
        2 1.0  +. 1     increment
          0.0  -+ 3     this is a skip, decr and continue
        3 001  .. .     zero after decr, find end of skip instruction
          101  .- 1     end found, start over
          011  .+ .     count zeros until end of instruction
          111  .. 0     found end, start skipping
```

All 8 input symbols are used (maybe not in every state), the 6 output actions are `c. .. .- +. -+ .+`, so the basic TM index is $3 \cdot \sqrt{\frac{8 \cdot 6}{2}} = 3 \cdot \sqrt{24} = 14.7$. A cyclic programme tape and two more tapes behind the window give a penalty of 3, thus the estimated TM index is $3 \cdot \sqrt{\frac{8 \cdot (6+3)}{2}} = 3 \cdot \sqrt{36} = 18.0$.

As is easily seen, states 2 and 3 may be joined, giving:

```
S PRJ  rj s    Comments
1 1.0  c. .    cycle instruction
  0.0  .. 2    incr or skip
  111  .- .    skip ones and count
  011  .. .    skip zeros
2 1.0  +. 1    increment
  0.0  -+ .    this is a skip, decr and continue
  001  .. .    zero after decr, find end of skip instruction
  101  .- 1    end found, start over
  011  .+ .    count zeros until end of instruction
  111  .. 0    found end, start skipping
```

So the basic TM index is $2 \cdot \sqrt{\frac{8 \cdot 6}{2}} = 2 \cdot \sqrt{24} = 9.8$, and the estimated TM index is $2 \cdot \sqrt{\frac{8 \cdot (6+3)}{2}} = 2 \cdot \sqrt{36} = 12.0$, which are rather remarkable, finally justifying Hasenjaeger's ideas.

3.5. Changing loop order

As indicated above, jumping if a register is not zero is tempting to make loops easy, but mastering border conditions is difficult. If, on the other hand, jump on zero would have been provided, the unconditional jump to loop begin is required nearly for each other jump, blowing up code size.

So Minsky's solution is preferable, but either there would be another instruction, or always a zero register available for the unconditional jump. In particular if the cycle instructions are efficient, a three tape machine can have conceptually a fourth tape, that is really non-writing, i.e. even cannot move its head. As the examples show, the registers are cycled through in each loop repetition anyhow, so its not a problem to cycle one step more at the end of the loop to have the zero register selected, then jump back, and then change to the first register really demanded. However, the handy contraction of decrement and jump conditional is no longer available.

A not yet studied solution might be to make the jump instruction dependent:

- if the instruction executed before was cycle, jump is when the register is zero.
- otherwise, the jump is unconditional.

3.6. Open Questions

My primary question would be to understand how Jones-Matysecivc Masking is applied to the given machine, and what changes with the variants presented here.

Apart from that, a major question arises if compared to the sentence on top of page 251:

> *If the jump condition is satisfied but instead of f...fd the next sign is c the evaluation of this situation in the sense of Boolean algebra II is to connect it with the top of A*

However, from the state table, there is no difference (except the comment) in the state table.

The minor questions of the state table interpretation have already been covered above:

- What if a condition is not covered, like a `d` if the register is zero, or the chain of `f`s is terminated by `e`?
- Was it intended that the first `f` starting the chain is not counted, and the the letter stopping a chain is not executed?

4. Materializing a register machine with non-writing tapes

Different materializations of the above concepts are studied with some examples, and more variants developped.

4.1. Cyclic tape, forward jumps only

Building a machine according to the above state table(s) is not a real problem, provided that a nice example programme is found in order to determine the length of the tapes. Using a cyclic tape with forward jumps will make the machine rather slow for large programmes, as copying a register of value *n* to another one requires *n* times rotating through the programme tape. On the other hand, visualising this behaviour, the machine has a certain appeal as the programme tape continuously rotates, giving the feeling of a clockwork, and could be related by a guide to the clock in a modern processor.

4.1.4. Original skip numbers

The examples use skipping all non-`f` instructions as in the article, and halting if a `d` is preceeded by a fall-through `f`.

To subtract the second (non-zero) from the first register would read, using `f d` on zero register as a stop:

$$\texttt{c\ \ c c d c d f}^{2}\texttt{ d}$$

Note that there is a cycle before entry into the loop, as the loop assumes register 2 to be active, so the skip count is 2.

Adding the two first (non-zero) registers to the third gives:

$$\texttt{c c e c d f}^{7}\texttt{\ \ c\ \ c e c c d f}^{7}\texttt{ d}$$

If the input registers may be zero, they are each incremented first, and the result decremented twice, as Hasenjaeger's instructions do not skip the loop if the control register is zero:

$$\texttt{e e\ \ c c e c d f}^{13}\texttt{\ \ c\ \ c e c c d f}^{13}\texttt{ c d d\ \ c f}^{1}\texttt{\ \ d}$$

Note the stop is achieved by `c f d`, i.e. by changing to register 1 again, which is zero from the first loop.

4.1.5. Skip cycle instructions only

Observing loops, it can be seen that a loop nearly always lands on a `c` instruction. This is not surprisingly, as a register has just been decremented, and the next action will in general modify another register. If not, a redundant series of cycle instructions must be used.

Thus, the `f` chains could be made shorter if the number of `f` instructions denote the number of `c` instructions to be skipped. `e`, `d` and `f` instructions are just skipped without being counted, even those following the last skipped `c`. As above, `f d` is a halt when the `d` is executed because the current register is zero and the jump was not taken.

The example to subtract the second (non-zero) from the first register with a halt reads now:

$$\texttt{c}\quad \texttt{c c d c d f}^1\ \texttt{d}$$

Note that a single `d` is sufficient for the stop, as the loop register is zero at end.

Adding the two first (non-zero) registers to the third now has 6 programming positions less:

$$\texttt{c c e c d f}^4\quad \texttt{c}\quad \texttt{c e c c d f}^4\ \texttt{d}$$

If the input registers may be zero, they are incremented first, and the result decremented twice:

$$\texttt{e e}\quad \texttt{c c e c d f}^6\quad \texttt{c}\quad \texttt{c e c c d f}^6\quad \texttt{c d d c f}^1\quad \texttt{d}$$

Here, the savings are $45 - 31 = 14$ instruction, about 30%.

However, it seems that the state table requires an additional state to skip the instructions following the last `c` skipped.

4.2. Splitting forward and backward jumps

While a unidirectional programme tape has its merits, once a bidirectional stepping mechanism is available, it could applied to the register tapes as well as to the programme tape.

So the questions to be tackled in this section are:

- how could we support backward as well as forward jumps?
- can we save one instruction letter using *conditional decrement and jump*?

One answer to the latter question was already shown in 3.4.

4.2.1. First Variant: skip operations and provide backward jump

To support backward jumps, the number of instructions is increased:

- c: cycle register
- +: increment register (instead of *e*)
- -: decrement register (limited by zero)
- f: skip forward if register is zero
- b: skip backwards if register is not zero

We keep to the rule that a chain of jump letters denotes the jump distance, and the jump distance counts only operation letters, not jump letters; the variant to skip only `c` operations is not considered here, see below for better solutions. The tape is not (necessarily) cyclic, and the machine stops if the end of the tape is reached. Unless noted, three tapes are assumed. We write `3f` for `f f f` instead of `f`^{3}. Comments are added informally after some spacing.

Clearing a register then reads

```
f - b
```

The first `f` is not necessary, as the decrement is limited, but will be useful later. Note that some border cases, in particular zero loop variables on entry, are now easy to solve by a forward jump at the begin of the loop.

Adding the current register to the next:

```
5f c + c c - 5b
```

Here, the initial `5f` is necessary in case the current register is zero.

Note the small asymmetry that `b` goes back to the previous instruction and `f` skips the next instruction (instead of going to the next one); nevertheless the repeat counts are the same.

Add r1 to r2, restoring r1, r3 zero on entry, i.e. `(2+ 3+ 1-) (1+ 3-)`

```
6f c + c + c - 6b
c c 5f c c + c - 5b
```

Divide by 2 into second register, i.e. `1- ( 2+ 1- 1-)`:

```
- 5f c + c - - 5b
```

4.2.2. Second variant: use jump letters only

In the structured approach, each jump, forward or backward, always could go to the opposite kind of jumps. So we could skip to jump letters instead of operations:

- `f` means: if the register is zero, skip forward until `b`.
- `b` means: if the register is not zero, skip backwards until `f`.

Note that in both cases, the jump may either land on a complementary jump letter, which is skipped, as the condition is the inverse one, or the following instruction.

Clearing a register is the same again:

```
f - b
```

Now the jump letter `f` is needed as a jump target.

Add the current register to the next:

```
f c + c c - b
```

Add r1 to r2, restoring r1, r3 zero on entry, i.e. `(2+ 3+ 1-) (1+ 3-)`

```
f c + c + c - b
c c f c c + c - b
```

Divide by 2 into second register, i.e. `1- ( 2+ 1- 1-)`:

```
- f c + c c - - b
```

To multiply, four registers are needed, as the multiplicant has to be saved, i.e. `( ( 3+ 4+ 2-) ( 2+ 4-) 1-)`:

```
3f  c  f c + c + c c - b  c c  f c c + c c - b  c - 3b
```

4.2.3. Third variant: Paired jump letters

As already used by Carstensen, instead of counting the jump letters, their nesting could be counted, requiring no argument at all. Thus, we treat them as parenthesis, and use these characters instead:

- `(` means: if the current register is zero, skip forward to the corresponding `)`.
- `)` means: if the current register is not zero, skip back to the corresponding letter `(`.

It is clear that *corresponding* means of the same nesting level. As the programme tape is not cyclic, stop is end of tape; alternatively, the symbol pair `( )` repeats endlessly if the current register is not zero, and `+ ( )` does this always.

Clearing a register becomes

```
( - )
```

Add with recover becomes:

```
( c + c + c - )        move 1st to 2nd and 3rd
c c ( c c + c - )      move 3rd to 1st
```

Divide by 2 into second register, i.e. `1- ( 2+ 1- 1-)`:

```
- ( c + c c - - )
```

Multiplication with 4 registers, equivalent to `( ( 3+ 4+ 2-) ( 2+ 4-) 1-)`:

```
( c ( c + c + c c - ) c c ( c c + c c - ) c - )
```

Using a programme tape with five symbols, namely `(c+-)`, the state table is rather straightforward using 3 states:

```
S PQR pqr S'
                State 0: sequential instructions
0 c.. +.c .     cycle
0 +.. +.+ .     increment
0 -.0 +.. .     no decrement, already zero
0 -.1 +-. .     decrement
0 (.0 +.. 1     reg is zero, skip forwards
0 (.1 +.. .     reg is not zero, proceed
0 ).0 +.. .     reg is zero, proceed
0 ).1 -.. 2     reg is not zero, repeat back
                State 1: skip forward
1 c.. +.. .     skip cycle
1 +.. +.. .     skip increment
1 -.. +.. .     skip decrement
1 (.. ++. .     push new loop
1 )0. ... 0     corresponding loop end found
1 )1. +-. .     pop loop end
                State 2: skip backwards
2 c.. -.. .     skip cycle
2 +.. -.. .     skip increment
2 -.. -.. .     skip decrement
2 (0. -.. 0     corresponding loop begin found
2 (1. --. .     inner loop level down
2 ).. -+. .     inner loop level up
```

For the TM index, all $5 \cdot 2 \cdot 2 = 20$ input symbols are used, the 10 output symbols are `+.c` `+.+` `+.-` `+..` `-..` `++.` `...` `+-.` `-..` `-+.` giving a basic TM index of

$3 \cdot \sqrt{20 \cdot \frac{10}{2}} = 3 \cdot \sqrt{100} = 30$.

Trying to reduce the TM index by eliminating unused input combinations, as `Q` is always zero in state 0, and `R` always 1 in states 1 and 2, the table could be written more restrictive:

```
S PQR pqr S'
0 c0. +.c .     cycle
0 +0. +.+ .     increment
0 -0. +.- .     decrement
0 (00 +.. 1     reg is zero, skip forwards
0 (01 +.. .     reg is not zero, proceed
0 )00 +.. .     reg is zero, proceed
0 )01 -.. 2     reg is not zero, repeat back
1 c.1 +.. .     skip cycle
1 +.1 +.. .     skip increment
1 -.1 +.. .     skip decrement
1 (.1 ++. .     push new loop
1 )01 ... 0     corresponding loop end found
1 )11 +-. .     pop loop end
```

```
2 c.1 -.. .     skip cycle
2 +.1 -.. .     skip increment
2 -.1 -.. .     skip decrement
2 (01 -.. 0     corresponding loop begin found
2 (11 --. .     inner loop level down
2 ).1 -+. .     inner loop level up
```

From the 20 possible input symbols, five are not used: `c10, +10, -10, (10, )10`, thus the basic TM index is $3 \cdot \sqrt{15 \cdot \frac{10}{2}} = 3 \cdot \sqrt{75} = 25.9$.

4.2.4. Forth variant: Paired jump letters with auto-decrement

As any backward jump is normally preceeded by decrementing the loop index register, the semantics of the closing bracket could be changed to:

- `)`: if the current register is not zero, decrement and skip back to the corresponding letter `(`. To indicate this, curly braces, `{` and `}`, are used instead.

Clearing a register becomes

```
{ }
```

Add with recover becomes:

```
{ c + c + c }
c c { c c + c }
```

Divide by 2 into second register, i.e. `1+ ( 2+ 1- 1-)`:

```
+ { c + c c - }
```

Multiplication with 4 registers needs 21 instead of 24 symbols:

```
{ c { c + c + c c } c c { c c + c c } c }
```

The state table is essentially the same, only this line changed:

```
S PQR pqr S'
0 }.1 -.- 2     reg is not zero, decrement and repeat back
```

The gain in space on the progamming tape is no really significant justifying this variant.

Note that a way to replace the single decrement by a construct using the auto-decrement closing brace has not yet been found.

4.3. Programme encoding

4.3.1. Direct encoding

Direct encoding with a fixed number of bits per symbols has practical advances; in particular, as Carstensen showed, to line up the symbols in a tray like holder for easy compilation of programmes.

Direct encoding of the five letters requires three bits, leaving three signs unused, of which one, the blank tape, could be used to stop the machine.

If, however, there are 8 symbols available anyhow, the *cycle* could be substituted by *select*, keeping the idea of a current window:

```
1-4     select register 1-4
+       increment
-       decrement
(       loop begin, skip to loop end if register is zero
)       loop end, skip back to loop begin if register is not zero
```

To halt the machines then, two options are:

- use an extra loop end without a corresponding loop start at the end. This means that at the end the tape goes back to the start in search of the corresponding opening parenthesis, and should stop there, as it is not cyclic.
- use the pair `()`, which would, if the current register is zero, do nothing, or loop endlessly if it is not zero; it would, however, tremble at the end.

The symbols are nearly the same as already used in 2.2, with some redundant register changes dropped; close to Carstensen's notation, except the parenthesis have no parameter. Producing the sum of `r1` and `r2` in `r3` (already zero), requires 14 symbols or 42 bits:

```
1( 3+ 1- )      move r1 to r3
2( 3+ 2- )      add r2 to r3
```

A Multiplication without saving the first factor requires 21 symbols or 63 bits:

```
1(
  2( 3+ 4+ 2- ) add r2 to r3 and save in r4
  4( 2+ 4- )    restore r2 from r4
1- )
```

Adding a fifth register, however, would require 4 bits, but allow for 12 registers, or 10 registes and ignore the blank symbols and have an extra halt symbol.

A straightforward state table for this variant may be:

```
S PQR pqr s
1 1.. +.1 .     select tape
  2.. +.2 .
  3.. +.3 .
  4.. +.4 .
  +.. +.+ .     increment
  -.. +.- ..    decrement
  (.0 +.. 2     start skip if zero
  (.1 +.. .     zero, ignore
  ).0 +.. .     not zero, ignore
  ).1 -.. 3     start skip backwards
```

```
2 1.. +.. .     skip
  2.. +.. .
  3.. +.. .
  4.. +.. .
  +.. +.. .
  -.. +.. .
  (.. ++. .     push level
  )0. +.. 1     close found, skip done
  )1. +-. .     pop level
3 1.. -.. .     skip
  2.. -.. .
  3.. -.. .
  4.. -.. .
  +.. -.. .
  -.. -.. .
  (0. -. .      open found, skip done
  (1. --. 1     pop level
  ).. -+. .     push level
```

This is technically also rather easy to realize, provided that the stop is done when then programme tape tries to step over the start.

The TM index, however, is not appealing: All $8 \cdot 2 \cdot 2 = 32$ input symbols are used, and there are 13 operations, thus the basic TM index is $3 \cdot \sqrt{\frac{32 \cdot 13}{2}} = 3 \cdot \sqrt{416} = 61.2$.

4.3.2. Variable coding

The variable length coding that Hasenjaeger used for his Mini-Wang (see section 4.4 in [Glaschick2012]) has the advantage that the often used *cycle* instruction could be assigned the shortest code, e.g.:

```
1       c       cycle registers
01      (       loop begin
001     )       loop end
0001    +       increment
00001   -       decrement
```

Every instruction has at least one bit on, so that backward scanning can be done with a reasonable number of states.

When during forward processing five zeroes (blanks on the tape) in a row occur, end of tape is assumed, and the machine halted.

Clearing the current register is encoded in 10 instead of 9 bit, same as direct encoding, but see below to include the decrement into the loop end:

```
(  -     )
01 00001 001
```

Adding the current register to the next one (with 3 tapes) needs 17 instead of 24 bits:

```
(  c +    c c -     )
01 1 0001 1 1 00001 001
```

Multiplication using 4 tapes requires 53 (instead of 63) bits:

```
1( 2 (  3 +    4 +    2   -     )   4   (  2   +    4   -     )   1 -     )
(  c (  c +    c +    c c -     )   c c (  c c +    c c -     )   c -     )
01 1 01 1 0001 1 0001 1 1 00001 001 1 1 01 1 1 0001 1 1 00001 001 1 00001 001
```

The decrement was intentionally assigned the longest code, if it is included in the closing parenthesis, we use curly braces instead:

```
1       c       cycle registers
01      {       loop begin, jump forward if zero
001     }       loop end, decrement and jump backwards if not zero
0001    +       increment
00001   -       decrement
```

Clearing the current register uses now only 5 instead of 9 bits:

```
{  }
01 001
```

Adding the current register to the next one needs 12 bits:

```
{  c +    c c }
01 1 0001 1 1 001
```

Dividing the current register by 2, leaving the result in the next one, needs 21 bits (3 tapes):

```
-     {  c +    c c -     }
00001 01 1 0001 1 1 00001 01
```

Multiplication (4 tapes) requires 40 (instead of 63) bits:

```
1( 2 (   3 +    4 +    2   -)  4   (  2   +    4   -)  1 -)
{  c {   c +    c +    c c }   c c {  c c +    c c }   c }
01 1 001 1 0001 1 0001 1 1 001 1 1 01 1 1 0001 1 1 001 1 001
```

The downside of this encoding is the larger number of states to decode the instructions to the state space. When skipping forward and `000` is found, skipping can continue until a `1` is found and skipped; there is no need to detect malformed input. When skipping backwards, the relevant patterns are `1 0 1`, which is a opening brace, and `1 0 0 1`, which is a closing brace. As the last one read is the end of the previous instruction, one forward step is needed to synchronize rightly. While skipping forward, the corresponding closing brace is skipped; because it would be ignored anyhow, as the register is zero, processing can continue with the next instruction. While skipping backwards, the scan stops at the end of the instruction before the opening brace, so it is scanned again forwards, which does not matter, as the register is zero and ignored anyhow; there is no need for extra states to skip it.

A straightforward version without state expansion by `Q` would read:

```
 S PQR pqr s
 0 1.. +.c .    cycle
   0.. +.. 1    0.
 1 1.0 +.. 5    loop begin, reg is zero, start forward skip
   1.1 +.. 0    loop begin, reg is not zero, ignore
   0.. +.. 2    00.
 2 1.0 -.. 0    loop end, reg is zero, ignore
   1.1 +.- 9    loop end, reg is not zero, decr, start backward skip
   0.. +.. 3    000.
 3 1.. +.+ 0    increment
   0.. +.. 4    0000.
 4 1.0 +.. 0    decrement, suppressed because zero
   1.1 +.- 0    decrement
   0.. ... .    four zeros in chain, halt
 5 1.. +.. 5    forward skip: cycle, ignore
   0.. +.. 6    0.
 6 1.. ++. 5    01: open brace, incr nesting level
   0.. +.. 7    00.
 7 10. +.. 0    001: close brace, zero nesting, loop end
   11. +-. 5    001: close brace, decr nesting level
   0.. +.. 8    000.
 8 1.. +.. 5    end of instruction found, just continue
   0.. +.. .    ignore zeros until end of instruction
 9 0.. -.. .    back, look for end of previous instruction
   1.. -.. 10   end of previous instruction found
10 1.. -.. 9    11: start over in backwards scan
   0.. -.. 11   01
11 10. +.. 0    101: open brace, zero nesting, loop end
   11. .-. 9    101: open brace, decr nesting level
   0.. -.. 12   001
12 1.. -.. 9    1001: close brace: incr nesting level
   0.. -.. 9    else skip back until end of previous instruction
```

For the TM index, all $2 \cdot 2 \cdot 2 = 8$ input symbols are used, the 10 output symbols are `+.c` `+..` `-..` `+.-` `+.+` `...` `++.` `+-.` `-..` `.-.`, thus the TM index is $12 \cdot \sqrt{8 \cdot \frac{10}{2}} = 12 \cdot \sqrt{40} = 75.9$, but the programme tape is very compact.

5. More elaborate examples for register machines

In search for an interesting demonstrator application for a materialized machine, the following examples were found.

5.1. Fibonacci numbers

An indefinite series of Fibonacci numbers is generated by the formula

$$k_{n+1} = k_n + k_{n-1}$$

This can be done with three registers x, y and z, where:

- x is k_{n-1} (initially zero)
- y is k_n (initially one)
- z is k_{n+1} (initially zero)

performing:

- move x to z
- add y to z and copy it to x
- move z to y

Maybe some moves could be saved by using the registes cyclically, but this is left as an excercise to the author.

Using register numbers instead, the inner loop reads:

```
( 3+ 1-)
( 3+ 1+ 2-)
( 2+ 3-)
```

Adding a loop around gives:

```
(( 3+ 1-)
 ( 3+ 1+ 2-)
 ( 2+ 3-)
3+ 3)           repeat without decrement
```

Using machine in ???4.2.3, just the numbers have to be resolved by c instructions, requiring 27 symbols or 81 bits at 3 bits per symbol:

```
( ( c c + c - )
c ( c + c c - )
c ( c c + c - )
+ - )
```

The encoding for the compact Hasenjaeger machine from 3.4 does not need the outer loop, as the tape is cyclic, and thus needs 59 bits:

```
c c +  c s
1 1 01 1 0^14 1
c c +  c c s
1 1 01 1 1 0^11 1
c c c +  c s
1 1 1 01 1 0^11 1
```

The variable coding from ???4.3.2 requires 46 bits and will run quicker, as is goes back for the loops:

```
{  {  c c +    c }
01 01 1 1 0001 1 001
c {  c +    c c }
1 01 1 0001 1 1 001
c {  c c +    c }
1 01 1 1 0001 1 001
+    }
0001 001
```

If the counters are modulo (preferrably the same modulus), the machine will cylce. There are e.g. 15 numbers in the series modulo 10: `1 1 2 3 5 8 3 1 4 5 9 4 3 7 0` having a sum of 56. There are 3 loops decrementing the current number, so we have about 3*56 turns through the programme. If the latter has 48 bits, the number of clock cylces needed is 3*46*48=8064, at 5 Hz these are 1612 seconds, nearly 30 minutes.

5.2. Collatz series

While having a lot of names, the term *Collatz series* will be used here for

$$k_{n+1} = \frac{k_n}{2} \quad \text{if } k_n \text{ is even}$$
$$k_{n+1} = 3 \cdot k_n + 1 \text{ if } k_n \text{ is odd}$$

which for all probed inital numbers goes into the final loop

```
4 → 2 → 1
```

We will make use of the fact that in the second case, the result is always even, and thus we use the condensed series:

$$k_{n+1} = \frac{k_n}{2} \quad \text{if } k_n \text{ is even}$$
$$k_{n+1} = \frac{3 \cdot k_n + 1}{2} = k_n + \frac{k_n + 1}{2} \text{ if } k_n \text{ is odd}$$

As it is fairly difficult just to check whether a number is odd, but much more easy to determine quotient and remainder in parallel:

$$k_n = 2 \cdot q_n + r_n$$

if r=0: $k_{n+1} = q_n$

else: $k_{n+1} = \frac{3 \cdot (2 \cdot q_n + 1) + 1}{2} = 3 \cdot q_n + 2$

Lacking conditionals, a multiply with r_n is necesessary:

$$k_{n+1} = q_n + 2 \cdot r_n \cdot (q_n + 1)$$

The following programme requires 4 tapes:

```
(
 ( 2+ 3+ 1-)            start division by 2
 3-
 ( 1+ 4+ 2- 2- 3- 3-)   q in r1 and r4, r in r2
 4+                     q+1 in r4
 ( (3+ 4-) 2-)          r3 = r4 * r2 = r*(q+1)
 ( 4-)                  clear r4
 ( 4+ 2+ 3-)            prepare *2
 ( 2+ 3-)               r2 = 2*r*(q+1)
 ( 1+ 2-)               r1 = r1 + r2
)
```

Note that the multiplication by r could be written shorter, as the inner code is executed at most once, because r2 is either zero or one, so the r4 need not be saved.

5.3. Linear congruential pseudo-random numbers

Using the linear congruential method, pseudo-random numbers with a maximum sequence can be generated.

Using cyclic tapes for the registers, the modulus operation is done automatically, and D.E. Knuth gives the rules when the pair of multiplicand and addition gives maximum sequence.

For example, the formula

$$x_{n+1} = (x_n \cdot 5 + 1)\,\%\,16$$

generates the sequence `1 6 15 12 13 2 11 8 9 14 7 4 5 10 3 0`.

The core is fairly small and needs only two registers:

```
 ( 2+ 2+ 2+ 2+ 2+ 1-)   mult by 5
 2+                     add 1
 ( 1+ 2-)               copy back
```

As starting with 0 is possible and delivers 0 as the last value, the surrounding loop could iterate as long as the result is not zero:

```
(
 ( 2+ 2+ 2+ 2+ 2+ 1-)   mult by 5
 2+                     add 1
 1+                     add one more to protect next loop
 ( 1+ 2-)               copy back
1-)                     loop until result is zero
```

The machine code for 3 tapes and auto-decrement on loop end will be:

```
c + + + + + c c s(-8)
c + c + c
c c + c s(-4)
s(-16)
```

As the copy back will be wrong if the result is zero, the number is incremented by one before, this is needed for the surrounding loop anyhow.

Using the minimal machine in section 3.4, the coding will require 69 bits:

```
c +  +  +  +  +  c c s
1 01 01 01 01 01 1 1 00^14 1
c +  c  +  c
1 01 1  01 1
c c +  c s
1 1 01 1 00^18 1
s
00^3 1
c c c
1 1 1
```

Note that the embracing loop would require a *decrement and skip zero*, as the tape is cyclic, but at least one skip must be used, so the redundant `ccc` is used. As single `c` would just repeat the sequence using a different register.

The time to run before repetition is fairly large; each number is decremented to zero twice, using 69 clock pulses. As the sum of the numbers is 120, more than $2 \cdot 120 \cdot 69 = 4777560$ clock cylcles are needed before repeating, running for nearly 55 days with 1 Hz, and more than a week with 5 Hz. Using a single `c` at the end, the running time before repeat will be 3 times larger.

6. Universal Register Machines and Complexitiy considerations

The term *Universal (Turing) Machine* was coined by Alan Turing, when he observed that the state table of a TM could be encoded on the tape, and a universal machine could do the same thing as any TM, with the proper encoding of the tape. Comparing two Turing Machines is only sensible if both do the same task, so considering a universal machine is only logical.

Comparing Register Machines is more difficult, because the concept of a Universal Register Machine is not as prominent as for Turing Machines. The apperantly only publications with *Universal Register (or Minsky) Machine* in the title are [Korec1996] and [GregusovaKorec1979], plus a German Diploma thesis of limited use here, see [Dietz2000].

The reason here is not only that register machines are less prominent in research; the encoding of a state table using prime numbers leads to large and bulky universal machines, be it simulations of Turing or register machines.

Also, the convenient state-symbol-product for Turing machines needs to be extended for register machines, at least using the number of registers used as a factor.

Moreover, while Turing Machines are in general introduced using a state table, Register Machines are very often shown using flow diagrams or instruction chains, which are not convenient to compare, although it is not complicated to convert flow diagrams to instruction chains, and the latter to state tables.

Nevertheless, providing a universal register machine that has encoded a register machine in a register at start, apparently is orders of magnitude less efficient than the simulated machine, and no efficient solution is visible.

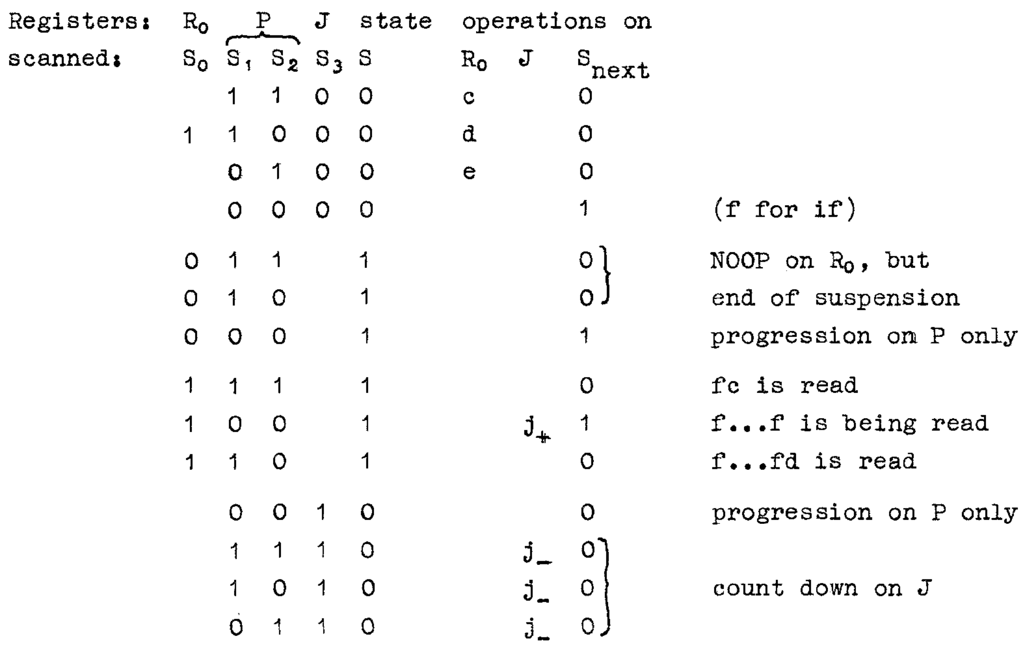
On the other hand, using Hasenjaeger's proposal to use Turing Machines to simulate register machines, the runtime seems to be in the same order of magnitude, i.e. in polynomial time, than the original register machine.

But as above it was shown that encoding a register machine definition with a fixed number of symbols is possible, thus the encoding could be in polynomial form, extracting the n-th symbol by a division by the location number. Or Minsky's technique could be used with factors instead of exponentiations.

Such a universal register machine will then have a better efficiency compared to the prime number encoding, maybe even polynomial time might be achievable directly, not via Hasenjaeger's simulation of a register machine on a Turing machine.

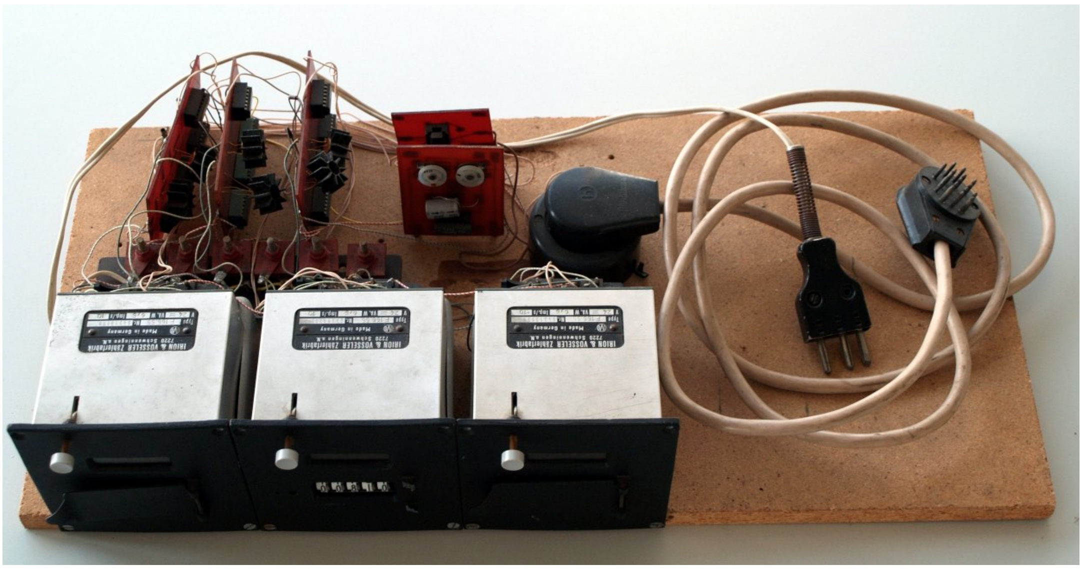
7. Appendix

7.1. Original state table



Registers:	R_0	P		J	state	operations on			
scanned:	S_0	S_1	S_2	S_3	S	R_0	J	S_{next}	
		1	1	0	0	c		0	
	1	1	0	0	0	d		0	
		0	1	0	0	e		0	
		0	0	0	0			1	(f for if)
	0	1	1		1			0	NOOP on R_0, but
	0	1	0		1			0	end of suspension
	0	0	0		1			1	progression on P only
	1	1	1		1			0	fc is read
	1	0	0		1		j_+	1	f...f is being read
	1	1	0		1			0	f...fd is read
		0	0	1	0			0	progression on P only
		1	1	1	0		j_-	0	
		1	0	1	0		j_-	0	count down on J
		0	1	1	0		j_-	0	

7.2. Triple counting device



8. References

Carstensen1975:
: Carsten Carstensen: *Eine schaltalgebraische Realisierung von Registermaschinen.* Flensburg (1975)

CohorsFresenborg1977:
: Elmar Cohors-Fresenborg: *Mathematik mit Kalkülen und Maschinen.* Vieweg Braunschweig (1977)

Dietz2000:
: Harry Hartmut Dietz: *Untersuchungen zum universellen Register-Maschinen-Programm*. Diplomarbeit FH Mannheim (2000). www.harrz.com/Diplom.pdf

Glaschick2012:
: Rainer Glaschick: *A size index for multitape Turing Machines*. Technical Report NI12061-SAS, Isaac Newton Institute for Mathematical Sciences, Cambridge, UK (2012)

GregusovaKorec1979:
: Ludmila Gregušová, Ivan Korec: *Small universal Minsky machines*. LNCS 74 (1979), pp. 308-316.

Hooper1969:
: Philip K. Hooper: *Some small, multitape universal Turing machines*. Information Sciences, vol. 1 no. 2, pp. 205 - 215 (1969).

Hasenjaeger1984:
: Gisbert Hasenjaeger: *Universal Turing Machines (UTM) and Jones-Matiyasevich-Masking*. In: E.Börger, G.Hasenjaeger, D.Rödding (Hrsg.): *Logic and Machines: Decisions Problems and Complexity*. Lecture Notes in Computer Science (1984) 248-253

Hasenjaeger1987:
: Gisbert Hasenjaeger: *On the early history of register machines*. in: E. Börger (Hrsg.): *Computation Theory and Logic*. Lecture Notes in Computer Science 270 (1987) 181-188

Korec1996:
: Ivan Korec: *Small universal register machines*. Theoretical Computer Science, vol. 168, no. 2, pp.267-301 (1996).

Minsky1961:
Marvin Minsky: *Recursive Unsolvability of Post's Problem of "TAG" And Other Topics in Theory of Turing Machines*. Annals of Mathematics, Vol. 74, No. 3 (November 1961)

Minsky1967:
Marvin Minsky: *Computation: Finite and Infinite Machines*. Prentice-Hall (1st ed., 1967)

Roedding1972:
Dieter Rödding: *Registermaschinen*. Der Mathematikunterricht, Band 18, S.32-41 (1972)

ShepherdsonSturgis1963:
J. C. Shepherdson, H.E. Sturgis: *Computability of Recursive Functions*. J. ACM, vol. 10 No 2 (1963), pp. 217-255.

---

[1]using real tapes, there is no quick way back to the beginning

[2]I am using the first edition; maybe this is changed in later editions

[3]which not only makes typing them easier, but for me also more readable.

[4]Roedding and Cohors-Fresenborg increment before the loop, and decrement the result.

[5]like in the second variant of the compact Mini-Wang in [Glaschick2012], where this column is still shown.