Proposal: "Alternative" RV128 vision for legacy 64b systems to transition directly to 128b & skip RV64 entirely

RV64 can be skipped entirely by legacy 64b systems & RV128 should *not* be an extrapolation from RV64

* My vision for RV128 sees it as a direct upgreade path for incumbent 64b systems (and there is little benefit in an RV64 intermediate transition)
* IMHO, the ingenuity of RISC V is the base (non RVC) ISA fits in 29-30 bits (<25%) of the 32b opcode space, leaving >75% for extensions.
* My Alternative RV128 takes this "ingenuity" further and is only 28 bits, or 6% (1/16th) of 32b opcode space & 94% is available for other uses.

Why make RV128 changes which will break "compatibility" with RV64?

* The divergence from RVI64 is minor, but required omitting the link register from the ABI (to allow x86_64 function call interoperability).
  I had to re-encode instruction formats. But a majority of the RISC V integer ISA is otherwise intact - register fields remain non destructive 5 bit.
  Toolchain/compiler changes (needed anyway for 128 bit) are only minimally increased compared to original RV128.
* "Original" RV128 is not binary compatible with RV64 anyway & RV64 has minimal installed base (in the target market of data centers & PC's).
  The advantages of RV64 retention are thus negligible & 100% RV64 consistency appears to be a design assumption not justified by hard data.
* My "Alternative" RV128 is likely to be more energy-efficient than "Original" RV128 (see below).
  The RV64 exposure of dual ALU ops sizes encourages energy efficient coding. I further extend this to ISA-exposed dual register sizes in RV128.
* By slimming down to 28b, the entire RV128 integer ISA can fit into a REX-like 4 bit prefix (in x86_64, REX has 16 x one byte opcodes = 1/16th of 256).
  My RV128 can thus be bolted-on as a low-overhead (& low transistor budget) datapath widening of a "host" x86_64 ISA.
  (This is an optional proprietary "extension", and my Alternative RV128 ISA also retains flexibility to use as a "greenfield" new CPU ISA).

I postulate extending all registers into full width 128 bits will be energy inefficient.

* Int32's will remain the standard int type even in a 128 bit world (eg. I32/LP64/LL128), as 2^32 is adequate for many non-pointer scalars
* Fused branches which always compare Int128's are higher energy consumption than comparing Int64's
* Many array indexes will likely remain 32 or 64 bits. (Pervasive ZEXT support is included where possible, but SEXT remains default)
* Transition to 128b pointer sizes will take time, and 64b pointer sizes will be adequate on 128b CPUs for many years (even decades).
  (eg. Linux X32 remained useful for 10+ yrs into 64 bit era, and WASM uses 32b address spaces even in a modern 64 bit OS/browser)

Alternative RV128 has 32 (16x64bit + 16x128bit) integer registers

* Register fields are non-destructive, and remain in a 5 bit format (so most instructions can mix 64b & 128b registers in any combination)
* For I32/L64/LLP128 & I32/LP64/LL128, this avoids the energy waste of writing 64 bits of SEXT/ZEXT & doing 128b branch-compares.
* The load/store addressing mode is base + 12 bit displacement (for word data sizes >= 32b)
* Initial "X64 ABI" will be useful for 1+ decades until 128b ABI takes over.
  Pointers remain 64b in this model, but data may be 128b:
    a1-a6 & t0-t1 are 64b arguments/temporaries
    r8-10,14-15 may be used for 128b arguments/temporaries
    s0, bp, sp & s1-s4 are 64b saved registers
    rz is hardwired zero (register 24 in 5 bit encoding)
    a0 is the return argument
  The above is intended to be interoperable with x86_64 ABIs
  See Long term ABI section (below) for 128b ABI design choices

64 bit	64 bit	128 bit	128 bit
0. a0	8. a5	16. r8	24. rz
1. a1	9. a6	17. r9	25. r1
2. a2	10. t0	18. r10	26. r2
3. s0	11. t1	19. r11	27. r3
4. sp	12. s1	20. r12	28. r4
5. bp	13. s2	21. r13	29. r5
6. a3	14. s3	22. r14	30. r6
7. a4	15. s4	23. r15	31. r7

* The base RV128 ISA proposed here has fused compare-branch & does not use condition codes.
  Condition codes may be supported via proprietary x86_64 extensions but are not part of the core Alternative RV128 spec.
  Note: x86_64 already has a conditional jump which compares a register against zero (no use CC use), so x86 already has a partial CC-free branch model
  Internally, x86_64 makes attempts to fuse CMP & Jcc instructions, and so already has the microarch needed for RV128 style fused branches.
* Procedure call/return will remain stack based (like x86_64), but there is scope to add link-register based jumps for "millicode" procedure calls

Instruction set and encoding

Continuous 5 bit register fields are shown below, but Alternative RV128 may opt to split & reshuffle register field bits (across all instructions)
For example moving the "wr" and "xd" bits in the diagrams below to the right side of "0100" will create a prefix byte of the form "0100.wrxd"
This byte is consistent with the x86_64 REX prefix (the x bit can be repurposed, as an index register is never used in Alternative RV128).
Aside from REX, the final design of Alternative RV128 may consider using other prefixes (eg. if dropping obsolete x87 FPU + 8 other x86 opcodes).
What is important at this stage is the entire integer ISA fits into 28b, and then it can be placed into any block of 16 (or 2x8) x86 instructions.
Also, host ISA embedding is just one option - the important ISA design principle here is flexibility over choice of "pure" or "mixed" ISA options.

Format RR: ALU_REG_REG

* xdddd, wrrrr, sssss are 5 bit register specifiers for destination, source1, source2 registers respectively
* zz=00 for ALU ops => then grouped by flag N into: N=1 for ALU_REG_REG (RR format), N=0 for ALU_REG_IMM (see RI format)
* zzff.f & yy specify these function codes:

** RR & RI formats **:		** RR format only **:			
ADD:	0000.0, yy=01	SUB:	yy=11	MUL:	yy=10
OR:	0000.1, yy=01	Reserved:	yy=11	MULH:	yy=10
AND:	0010.0, yy=01	ANDN:	yy=11	DIV:	yy=10
SLT:	0010.1, yy=01	PACK:	yy=11	REM:	yy=10
XOR:	0011.0, yy=01	XNOR:	yy=11	DIVU:	yy=10
SLTU:	0011.1, yy=01	PACKU:	yy=11	REMU:	yy=10
Reserved:	0001.0 [12b IMM]	MIN/MAX:	yy=01/11	MINU/MAXU:	yy=00/10
Shift/rotate:	0001.1 [All have RR format, or 7b IMM in YY.Sssss formats]:				
		SRL/SRA:	yy=01/11	ROR/SLL:	yy=00/10

* YYY=000/0--: SEXT (default), VV field selects data size as follows:
  VV = 11: dest & src data size set to full dest register size (ie: either 64b or 128b)
  VV = 01: dest & src data size set to half dest register size (ie: either 32b or 64b)
* YYY=100/1--: ZEXT, VV field selects data size (or selects MULHU/MULHSU variants of MULH):
  VV = 11: dest data size set to full (ie: 64/128b), src size set to half + ZEXT (ie: 32b/64b src2 [shifts src1])
  VV = 01: dest & src size set to half dest register size + ZEXT (ie: either 32b or 64b)

Format RI: ALU_REG_IMM (12b immediate), BRANCH (12b displacement), LOAD/STORE

* Y[8]+VV+yy: 12 bit signed immediate for ALU ops, 2 byte PC rel jump offset, or 4 byte signed offset for L/LW/ST/STW
  (This proposal defines immediate field sizes, but details of bit order inside the field are left out for now)
* ffff.f & N specifies these function codes:

ALU_I ops:	00ff.f, N=0	(see function codes in Format RR above)	
L (64/128b):	0100.0, N=0	LW (32b):	N=1
ST:	0100.1, N=0	STW:	N=1
Reserved:	1000.0, N=0/1	(reserved for x86 destructive ALU imm ops extension)	

```
Move:       1000.1, N=0              Move_ZEXT:   N=1
     LB/LH/LX (yy=01) or STB/STH/STX (yy=00) where B=8b (VV=00), H=16b (VV=10), X=32b/64b (VV=01, half register size)
     LEA (ie: ADDIW) if yy=10
     Y[8] = 8 bit offset/immediate & N selects SEXT (N=0) or ZEXT (N=1)
BEQ:        1001.0, N=0              BNE:        N=1
BLT:        1010.0, N=0              BGE:        N=1
BLTU:       1010.1, N=0              BGEU:       N=1
```

Format LI: LONG_IMMEDIATE


```
* 11ff.f & N specifies these function codes:
     LUI:     any 11ff.f, N=0          Y[8].YY.YYYYY + ff.fyy is 20b upper immediate value
     Reserved:   1100.0, N=1           (reserved for x86 destructive shifts extension)
     JAL:        1100.1, N=1           Y[8].YY.YYYYY + yy + ZZ + '0' is 20b displacement, aligned to even bytes
     Reserved:   1101.0, N=1           (reserved for x86 destructive shifts extension)
     Reserved:   1101.1, N=1           (potential JALR opcode)
     ADDUIPC:    1110.f, N=1           Y[8].YY.YYYYY + fyy + '00' is 20b upper immediate (LSB 2 bits = 00)
                                       Note: because L/LW/S/STW specify offsets scaled by 4, their range is 2^14 bytes
* ZZddd is destination register, except JAL (which currently doesn't write to any register)
* JAL restricted to ddd=0x0 (no ret addr) or ddd=0x7 (ret addr pushed to stack).  [Option to allow ret addr in r1-r6 being considered.]
```

Long Term 128b ABI and Design Choice of 24 vs 32 Registers

To support a future 128b ABI, a design option to consider is the register model shown below, with 24 true registers in 8-31, and psuedo-registers in 0-7.
The pseudo-registers are a 64 bit window into true 128b registers 16-23. Any writes into psuedo-registers are SEXT writes into the true register.
This option has 8 fewer registers, but may allow better transition to 128b pointers and ABI interop (ie: 64b ABI calling 128b ABI and vice versa)
64b ABI code can continue using a0-a4 (which are now psuedoregisters) for argument passing, and 128b code can still find 128b arguments in r8-10,14,15.
To avoid ambiguity, this 24 register design choice, if selected, will apply to all Alternative RV128 CPUs, regardless of 64b or 128b ABI and will not be a "fork".

64b (psuedo)	64 bit	128 bit	128 bit
0. a0'	8. a5	16. a0 (r8)	24. rz
1. a1'	9. a6	17. a1 (r9)	25. r1
2. a2'	10. t0	18. a2 (r10)	26. r2
3. s0'	11. t1	19. s0 (r11)	27. r3
4. sp'	12. s1	20. sp (r12)	28. r4
5. bp'	13. s2	21. bp (r13)	29. r5
6. a3'	14. s3	22. a3 (r14)	30. r6
7. a4'	15. s4	23. a4 (r15)	31. r7

X86_64 Host ISA

1. RV128 may be bolted on as a low-overhead addition to the x86_64 ISA, with low instruction decoder footprint, and re-using existing (widened) x86_64 ALU pathways
2. In such a scenario, legacy 8086 (real) and/or 286/386 (protected) modes would be dropped, and x86 systems would offer primarily x86_RV128 mode & x86_64 Long mode.
3. x86_RV128 mode would continue to use the x86_64 host ISA for shorter (2 byte) instructions, XMM (reusing SIMD execution pathways) & privileged instructions.
   (In original RV128, all of these would have to be redesigned from scratch, as their instruction opcodes would clash with 30 bit RV128 opcodes.)
4. Contrary to popular perception, one byte x86_64 instructions are not frequently used (mainly PUSH, POP, RET).
   They can be fused into pairs (eg. PUSH x 2 or POP x 2) or be followed by 1 byte NOP (eg. RET) & all functions are to be aligned to 2 byte start addresses.
5. Other implementations might be "pure" RV128, with no co-mingling of x86 instructions & would need RV128 extensions for FP, vector and privileged ops.